
Trusted Analytics Documentation

Release 0.4.0

Author: Intel

September 16, 2015

I	Technical Summary	1
1	Overview	5
2	Python and Data Frame User Interface Summary	7
3	Graph Pipeline Summary	9
4	Graph Analytics Summary	11
5	Machine Learning Summary	13
6	Plugins Summary	15
II	User Manual	17
7	Getting Started	19
7.1	Open-Source	19
7.2	Features	19
7.3	Script Examples	19
8	Process Flow Examples	21
8.1	Python Path Setup	21
8.2	Raw Data	22
8.3	Frames	23
8.4	Seamless Graph	33
8.5	Titan Graph	34
8.6	Graph Analytics	35
9	Machine Learning	39
9.1	Algorithms	39
9.2	Supervision	40
9.3	Other Resources	40
10	Best Known Methods (User)	41
10.1	Python	41
III	Extending Trusted Analytics	45
11	Tracking Metadata	47
11.1	Using PostgreSQL	47

11.2	Basic PSQL commands	49
12	IPython Setup Instructions	51
12.1	Introduction	51
12.2	Requirements	51
12.3	Configure Trusted Analytics Python REST Client	51
12.4	Running IPython	52
13	Plugin Authoring Guide	55
13.1	Introduction	55
13.2	Types of Plugins	56
13.3	When to Write a Plugin	56
13.4	Plugin Support Services	56
13.5	Creating a CommandPlugin	58
13.6	Creating an Archive	59
13.7	Deployment	59
13.8	Configuration	59
13.9	Archive Declaration	59
13.10	Enabling the Archive	60
14	Best Known Methods (Development)	61
14.1	Configuration information	61
IV	Administration	79
15	System Requirements	81
15.1	Physical Machine Configuration	81
15.2	Cloudera Hadoop 5 Configuration	83
16	Package Installation	95
16.1	Introduction	95
16.2	Requirements	96
16.3	Trusted Analytics Package Installation	96
16.4	REST Server Configuration	99
17	Scoring Engine	109
17.1	Installation	109
17.2	Scoring Models Implementation	109
17.3	Configuration of the Engine	109
17.4	Starting the Scoring Engine Service	110
17.5	Scoring Client	110
18	Virtual Machines	111
18.1	Introduction	111
18.2	Requirements	111
18.3	Download VM Image	112
18.4	Windows GUI Client	112
18.5	Extract Archive	116
18.6	Import Image	119
18.7	Running Trusted Analytics VM Image	119
18.8	Eclipse/PyDev	123
18.9	Logs	123
18.10	Updating	125
18.11	Common VM problems	126

19 Best Known Methods (Admin)	127
19.1 Configuration information	127
20 Windows Python Setup	145
21 Integrated Development Environments	147
21.1 Eclipse Setup Instructions	147
21.2 IntelliJ IDEA Setup Instructions	149
21.3 PyCharm Setup Instructions	151
 V Python API	 153
22 Connect to the Server	155
22.1 Basic connecting	155
22.2 Connections requiring OAuth	156
22.3 Using Environmental Variables	156
22.4 Troubleshooting	157
23 Data Types	159
24 Data Sources	161
24.1 CsvFile	161
24.2 HiveQuery	163
24.3 HBase	164
24.4 Jdbc	165
24.5 JsonFile	166
24.6 LineFile	167
24.7 Pandas	168
24.8 XmlFile	170
25 Frames	173
25.1 <i>Frames</i> EdgeFrame	173
25.2 <i>Frames</i> VertexFrame	211
25.3 <i>Frames</i> Frame	247
25.4 <i>trustedanalytics</i> drop_frames	289
25.5 <i>trustedanalytics</i> get_frame	289
25.6 <i>trustedanalytics</i> get_frame_names	290
26 Graphs	291
26.1 <i>Graphs</i> Graph	291
26.2 <i>Graphs</i> TitanGraph	304
26.3 <i>trustedanalytics</i> get_graph	314
26.4 <i>trustedanalytics</i> drop_graphs	315
26.5 <i>trustedanalytics</i> get_graph_names	315
27 Models	317
27.1 <i>Models</i> LibsvmModel	317
27.2 <i>Models</i> RandomForestClassifierModel	321
27.3 <i>Models</i> PrincipalComponentsModel	324
27.4 <i>Models</i> CollaborativeFilteringModel	327
27.5 <i>Models</i> KMeansModel	338
27.6 <i>Models</i> SvmModel	341
27.7 <i>Models</i> LdaModel	344
27.8 <i>Models</i> LogisticRegressionModel	358

27.9	<i>Models</i> NaiveBayesModel	362
27.10	<i>Models</i> LinearRegressionModel	364
27.11	<i>Models</i> RandomForestRegressorModel	367
27.12	<i>trustedanalytics</i> get_model	369
27.13	<i>trustedanalytics</i> drop_models	370
27.14	<i>trustedanalytics</i> get_model_names	370

VI REST API 371

28	<i>REST API</i> Commands	373
28.1	<i>Commands</i> Issue Command	373
28.2	<i>Commands</i> Get Command	374
28.3	<i>Commands</i> About Command Names	376
28.4	<i>Commands</i> _admin:/_explicit_garbage_collection	377
28.5	<i>Commands</i> frame:/filter	379
28.6	<i>Commands</i> frame:/join	380
28.7	<i>Commands</i> frame:/label_propagation	382
28.8	<i>Commands</i> frame:/load	385
28.9	<i>Commands</i> frame:/loopy_belief_propagation	387
28.10	<i>Commands</i> frame:/rename_columns	391
28.11	<i>Commands</i> frame:edge/add_edges	393
28.12	<i>Commands</i> frame:edge/rename_columns	394
28.13	<i>Commands</i> frame:vertex/add_vertices	396
28.14	<i>Commands</i> frame:vertex/drop_duplicates	398
28.15	<i>Commands</i> frame:vertex/filter	399
28.16	<i>Commands</i> frame:vertex/rename_columns	401
28.17	<i>Commands</i> frame/_coalesce	403
28.18	<i>Commands</i> frame/_partition_count	404
28.19	<i>Commands</i> frame/_repartition	406
28.20	<i>Commands</i> frame/_size_on_disk	407
28.21	<i>Commands</i> frame/add_columns	409
28.22	<i>Commands</i> frame/assign_sample	411
28.23	<i>Commands</i> frame/bin_column	413
28.24	<i>Commands</i> frame/bin_column_equal_depth	415
28.25	<i>Commands</i> frame/bin_column_equal_width	417
28.26	<i>Commands</i> frame/categorical_summary	419
28.27	<i>Commands</i> frame/classification_metrics	421
28.28	<i>Commands</i> frame/column_median	423
28.29	<i>Commands</i> frame/column_mode	425
28.30	<i>Commands</i> frame/column_summary_statistics	427
28.31	<i>Commands</i> frame/compute_misplaced_score	430
28.32	<i>Commands</i> frame/copy	432
28.33	<i>Commands</i> frame/correlation	434
28.34	<i>Commands</i> frame/correlation_matrix	435
28.35	<i>Commands</i> frame/count_where	437
28.36	<i>Commands</i> frame/covariance	439
28.37	<i>Commands</i> frame/covariance_matrix	440
28.38	<i>Commands</i> frame/cumulative_percent	442
28.39	<i>Commands</i> frame/cumulative_sum	444
28.40	<i>Commands</i> frame/dot_product	446
28.41	<i>Commands</i> frame/drop_columns	448
28.42	<i>Commands</i> frame/drop_duplicates	449
28.43	<i>Commands</i> frame/ecdf	451

28.44	<i>Commands</i> frame/entropy	453
28.45	<i>Commands</i> frame/export_to_csv	454
28.46	<i>Commands</i> frame/export_to_hbase	456
28.47	<i>Commands</i> frame/export_to_hive	458
28.48	<i>Commands</i> frame/export_to_jdbc	460
28.49	<i>Commands</i> frame/export_to_json	461
28.50	<i>Commands</i> frame/flatten_column	463
28.51	<i>Commands</i> frame/group_by	465
28.52	<i>Commands</i> frame/histogram	466
28.53	<i>Commands</i> frame/loadhbase	468
28.54	<i>Commands</i> frame/loadhive	470
28.55	<i>Commands</i> frame/loadjdbc	472
28.56	<i>Commands</i> frame/quantiles	474
28.57	<i>Commands</i> frame/rename	475
28.58	<i>Commands</i> frame/sort	477
28.59	<i>Commands</i> frame/sorted_k	478
28.60	<i>Commands</i> frame/tally	481
28.61	<i>Commands</i> frame/tally_percent	482
28.62	<i>Commands</i> frame/top_k	484
28.63	<i>Commands</i> frame/unflatten_column	486
28.64	<i>Commands</i> graph:/_info	487
28.65	<i>Commands</i> graph:/define_edge_type	489
28.66	<i>Commands</i> graph:/define_vertex_type	491
28.67	<i>Commands</i> graph:/edge_count	492
28.68	<i>Commands</i> graph:/export_to_titan	494
28.69	<i>Commands</i> graph:/ml/kclique_percolation	495
28.70	<i>Commands</i> graph:/vertex_count	499
28.71	<i>Commands</i> graph:titan/export_to_graph	500
28.72	<i>Commands</i> graph:titan/graph_clustering	502
28.73	<i>Commands</i> graph:titan/query/gremlin	503
28.74	<i>Commands</i> graph:titan/vertex_sample	505
28.75	<i>Commands</i> graph/annotate_degrees	507
28.76	<i>Commands</i> graph/annotate_weighted_degrees	509
28.77	<i>Commands</i> graph/clustering_coefficient	512
28.78	<i>Commands</i> graph/copy	514
28.79	<i>Commands</i> graph/graphx_connected_components	516
28.80	<i>Commands</i> graph/graphx_pagerank	518
28.81	<i>Commands</i> graph/graphx_triangle_count	521
28.82	<i>Commands</i> graph/ml/belief_propagation	522
28.83	<i>Commands</i> graph/rename	524
28.84	<i>Commands</i> model:collaborative_filtering/new	526
28.85	<i>Commands</i> model:collaborative_filtering/recommend	530
28.86	<i>Commands</i> model:collaborative_filtering/train	532
28.87	<i>Commands</i> model:k_means/new	534
28.88	<i>Commands</i> model:k_means/predict	536
28.89	<i>Commands</i> model:k_means/publish	538
28.90	<i>Commands</i> model:k_means/train	539
28.91	<i>Commands</i> model:lda/new	541
28.92	<i>Commands</i> model:lda/predict	547
28.93	<i>Commands</i> model:lda/publish	549
28.94	<i>Commands</i> model:lda/train	550
28.95	<i>Commands</i> model:libsvm/new	553
28.96	<i>Commands</i> model:libsvm/predict	554
28.97	<i>Commands</i> model:libsvm/publish	556

28.98	<i>Commands</i> model:libsvm/score	558
28.99	<i>Commands</i> model:libsvm/test	559
28.100	<i>Commands</i> model:libsvm/train	561
28.101	<i>Commands</i> model:linear_regression/new	564
28.102	<i>Commands</i> model:linear_regression/predict	565
28.103	<i>Commands</i> model:linear_regression/train	567
28.104	<i>Commands</i> model:logistic_regression/new	569
28.105	<i>Commands</i> model:logistic_regression/predict	571
28.106	<i>Commands</i> model:logistic_regression/test	572
28.107	<i>Commands</i> model:logistic_regression/train	574
28.108	<i>Commands</i> model:naive_bayes/new	577
28.109	<i>Commands</i> model:naive_bayes/predict	579
28.110	<i>Commands</i> model:naive_bayes/train	580
28.111	<i>Commands</i> model:principal_components/new	582
28.112	<i>Commands</i> model:principal_components/predict	584
28.113	<i>Commands</i> model:principal_components/publish	586
28.114	<i>Commands</i> model:principal_components/train	587
28.115	<i>Commands</i> model:random_forest_classifier/new	589
28.116	<i>Commands</i> model:random_forest_classifier/predict	591
28.117	<i>Commands</i> model:random_forest_classifier/publish	593
28.118	<i>Commands</i> model:random_forest_classifier/test	594
28.119	<i>Commands</i> model:random_forest_classifier/train	596
28.120	<i>Commands</i> model:random_forest_regressor/new	598
28.121	<i>Commands</i> model:random_forest_regressor/predict	600
28.122	<i>Commands</i> model:random_forest_regressor/publish	602
28.123	<i>Commands</i> model:random_forest_regressor/train	603
28.124	<i>Commands</i> model:svm/new	605
28.125	<i>Commands</i> model:svm/predict	607
28.126	<i>Commands</i> model:svm/test	609
28.127	<i>Commands</i> model:svm/train	611
28.128	<i>Commands</i> model/rename	613
28.129	<i>Command List</i>	614
29	<i>REST API Entities</i>	619
29.1	<i>Entities</i> Create Entity	619
29.2	<i>Entities</i> Drop Entity	620
29.3	<i>Entities</i> Get Entity	621
29.4	<i>Entities</i> Get Named Entities	623
29.5	<i>Entities</i> Get Frame Data	624
30	<i>REST API Info</i>	627
30.1	GET /info	627
VII	References	629
31	Glossary	633
32	Legal Statement	645
33	Index	647
34	Appendices	649
34.1	Appendix A — Sample Application Configuration File	649

35 Errata	655
Bibliography	657
Index	659

Part I

Technical Summary

Table of Contents

- *Overview*
- *Python and Data Frame User Interface Summary*
- *Graph Pipeline Summary*
- *Graph Analytics Summary*
- *Machine Learning Summary*
- *Plugins Summary*

OVERVIEW

Trusted Analytics is a platform that simplifies applying *graph analytics* and *machine learning* to big data for superior knowledge discovery and predictive modeling across a wide variety of use cases and solutions. Trusted Analytics provides an analytics pipeline (ATK) spanning feature engineering, graph construction, graph analytics, and machine learning using an extensible, modular framework. By unifying graph and entity-based machine learning, machine learning developers can incorporate an entity's nearby relationships to yield superior predictive models that better represent the contextual information in the data. All functionality operates at full scale, yet are accessed using a higher level Python data science programming abstraction to significantly ease the complexity of cluster computing and parallel processing. The platform is fully extensible through a plugin architecture that allows incorporating the full range of analytics and machine learning for any solution need in a unified workflow that frees the researchers from the overhead of understanding, integrating, and inefficiently iterating across a diversity of formats and interfaces.

PYTHON AND DATA FRAME USER INTERFACE SUMMARY

ATK utilizes Python data science abstractions to make programming fully scalable big data analytic workflows using Spark/Hadoop clusters as familiar and accessible as using popular desktop machine learning solutions such as Pandas and SciKit Learn. The scalable data frame representation is more familiar and intuitive to data researchers compared to low level HDFS file and Spark RDD formats. ATK provides an extensive library to manipulate the data frames for feature engineering and exploration, such as joins and aggregations. User-defined transformations and filters can be written in Python and applied to terabytes (and more) of data using distributed processing. Machine learning algorithms are also invoked as higher-level data science API (Application Programming Interface) abstractions, making model development (such as creating parallel recommender systems or training classifier and clustering models) accessible to a broad population of researchers possessing mainstream data science programming skills. For more information, see the section on [process flow](#) and the [Python website](#)¹.

¹<http://www.python.org>

GRAPH PIPELINE SUMMARY

In addition to enabling use of entity-based data representations and algorithms, the toolkit provides a full graph pipeline to enable application of graph methods to big data. Graph representations are broadly useful, for example to link disparate data using arbitrary edge types, and then analyze the connections for powerful predictive signals that can otherwise be missed with entity-based methods. Working with graph representations can often be more intuitive and computationally efficient for data sets where the connections between data observations are more numerous and more important than the data points alone. ATK offers a representation of graph data as fully-scalable property graph objects with vertices, edges, and associated properties. The pipeline brings together into one workflow all the capabilities to create and analyze graph objects, including engineering features, linking data, performing rich traversal queries, and applying graph-based algorithms. Because data scientists often need to iterate analysis using both graph and frame representations (for example, applying a clustering algorithm to a vertex list with features developed using graph analytics), ATK provides the seamless ability to move between both data representations.

GRAPH ANALYTICS SUMMARY

Fully-scalable graph analytic algorithms are provided for uncovering central influences and communities in the data set. This ability is useful for exploring the data, as well as for incorporating as machine learning features that incorporate the context of an entity in the graph, thus creating better, more predictive, machine learning results.

MACHINE LEARNING SUMMARY

The toolkit provides algorithms for supervised, unsupervised, and semi-supervised machine learning using both entity and graphical machine learning tools. Graph machine learning algorithms such as label propagation and loopy belief propagation, exploit the connections in the graph structure and provide powerful new methods of labeling or classifying graph data. Examples of other machine learning capabilities provided include recommender systems using alternating least squares and conjugate gradient descent, topic modeling using Latent Dirchelet Allocation, clustering using K-means, and classification using logistic regression. See the section on [machine learning](#) and the [API](#) for further information.

PLUGINS SUMMARY

In addition to the extensive set of capabilities provided, the platform is fully extensible using a plugin architecture. This allows developers to incorporate graph analytical tools into the existing range of machine learning abilities, expanding the capabilities of Trusted Analytics for new problem solutions. Plugins are developed using a thin Scala wrapper, and the ATK framework automatically generates a Python presentation for those added functions. Plug-ins can be used for a range of purposes, such as developing custom algorithms for specialized data types, building custom transformations for commonly used functions to get higher performance than a UDF (Python User-defined Function), or integrating other tools to further unify the workflow. See the [Plugin Authoring Guide](#) for more information.

Part II

User Manual

GETTING STARTED

Table of Contents

- *Open-Source*
- *Features*
- *Script Examples*

7.1 Open-Source

Trusted Analytics uses standards and open-source routines from [Apache Hadoop](http://hadoop.apache.org/)¹ such as HDFS (Hadoop Distributed File System), *MapReduce*, YARN, as well as [Apache Giraph](http://giraph.apache.org/)² for graph-based machine learning and graph analytics. The Titan graph database can be queried using the [Gremlin](https://github.com/tinkerpop/gremlin/wiki)³ graph query language from TinkerPop.

7.2 Features

- Import routines read and convert data from several different formats
- Data cleaning tools prepare the data by removing erroneous values, transforming values to a normalized state and constructing new features through manipulating existing values
- Analysis and machine learning algorithms give deeper insight into the data

7.3 Script Examples

Trusted Analytics ships with example Python scripts and data sets that exercise the various features of the platform. The default location for the example scripts is ta's home directory `/home/ta`.

The examples are located in `/home/trustedanalytics/examples`:

```
-rwxr-xr-- 1 atkuser atkuser 904 Jul 30 04:20 als.py
-rwxr-xr-- 1 atkuser atkuser 921 Jul 30 04:20 cgd.py
-rwxr-xr-- 1 atkuser atkuser 1078 Jul 30 04:20 lbp.py
-rwxr-xr-- 1 atkuser atkuser 707 Aug 7 18:21 lda.py
-rwxr-xr-- 1 atkuser atkuser 930 Jul 30 04:20 lp.py
```

¹<http://hadoop.apache.org/>

²<http://giraph.apache.org/>

³<https://github.com/tinkerpop/gremlin/wiki>

```
-rwxr-xr-- 1 atkuser atkuser 859 Jul 30 04:20 movie_graph_5mb.py
-rwxr-xr-- 1 atkuser atkuser 861 Jul 30 04:20 movie_graph_small.py
-rwxr-xr-- 1 atkuser atkuser 563 Jul 30 04:20 pr.py
```

The datasets are located in ‘/home/trustedanalytics/examples/datasets’ and ‘hdfs://user/trustedanalytics/datasets/’:

```
-rw-r--r-- ... /user/trustedanalytics/datasets/README
-rw-r--r-- ... /user/trustedanalytics/datasets/apl.csv
-rw-r--r-- ... /user/trustedanalytics/datasets/lbp_edge.csv
-rw-r--r-- ... /user/trustedanalytics/datasets/lp_edge.csv
-rw-r--r-- ... /user/trustedanalytics/datasets/movie_sample_data_5mb.csv
-rw-r--r-- ... /user/trustedanalytics/datasets/movie_sample_data_small.csv
-rw-r--r-- ... /user/trustedanalytics/datasets/recommendation_raw_input.csv
-rw-r--r-- ... /user/trustedanalytics/datasets/test_lda.csv
```

The datasets in ‘/home/trustedanalytics/examples/datasets’ are for reference. The actual data that is being used by the Python examples and the Trusted Analytics server is in the HDFS system.

To get access to the scripts, login as atkuser and go to the example scripts directory:

```
$ sudo su atkuser
$ cd /home/trustedanalytics/examples
```

To run any of the Python example scripts type:

```
$ python <SCRIPT_NAME>
```

where “<SCRIPT_NAME>” is any of the scripts.

PROCESS FLOW EXAMPLES

Table of Contents

- *Python Path Setup*
- *Raw Data*
 - *Ingesting the Raw Data*
 - * *Importing a CSV (Character-Separated Variables) file.*
- *Frames*
 - *Create A Frame*
 - * *Examples:*
 - * *Append:*
 - *Inspect The Data*
 - * *Examples*
 - *Clean The Data*
 - * *Drop Rows:*
 - *Examples:*
 - * *Filter Rows:*
 - *Examples:*
 - * *Drop Duplicates:*
 - *Examples:*
 - * *Drop Columns:*
 - * *Rename Columns:*
 - *Transform The Data*
 - * *Add Columns:*
 - *Examining the Data*
 - * *Group by (and aggregate):*
 - * *Join:*
 - * *Flatten Column:*
- *Seamless Graph*
 - *Build the Graph*
 - *Other Graph Options*
- *Titan Graph*
 - *Graph Creation*
- *Graph Analytics*

8.1 Python Path Setup

It is recommended that the location of the ‘trustedanalytics’ directory be added to the PYTHONPATH environmental variable prior to starting Python. This can be done from a shell script, like this:

```
PYTHONPATH=$PYTHONPATH:/usr/lib/  
export PYTHONPATH  
python
```

This way, from inside Python, it is easy to load and connect to the REST server:

```
>>> import trustedanalytics as ta  
>>> ta.connect()
```

8.2 Raw Data

Data is made up of variables of heterogeneous types (for example: strings, integers, and floats) that can be organized as a collection of rows and columns, similar to a table or spreadsheet. Each row corresponds to the data associated with one observation, and each column corresponds to a variable being observed. See the Python API [Data Types](#) for a current list of data types supported.

Connect to the server:

```
>>> import trustedanalytics as ta  
>>> ta.connect()
```

Note: Sometimes it is helpful to see the details of the python stack trace upon error. Setting the `show_details` to `True` causes the full python stack trace to be printed, rather than a friendlier digest.

```
>>> ta.errors.show_details = True
```

To see the data types supported:

```
>>> print ta.valid_data_types
```

You should see a list of variable types similar to this:

```
float32, float64, int32, int64, unicode  
(and aliases: float->float64, int->int32, long->int64, str->unicode)
```

Note: Although Trusted Analytics utilizes the NumPy package, NumPy values of positive infinity (`np.inf`), negative infinity (`-np.inf`) or nan (`np.nan`) are treated as `None`. Results of any user-defined functions which deal with such values are automatically converted to `None`, so any further usage of those data points should treat the values as `None`.

8.2.1 Ingesting the Raw Data

See the API section [Data Sources](#) for the various methods of ingesting data.

Importing a CSV file.

These are some rows from a CSV file:

```
"string",123,"again",25.125  
"next",5,"or not",1.0  
"fail",1,"again?",11.11
```


CSV files contain rows of information separated by new-line characters. Within each row, the data fields are separated from each other by some standard character. In the above example, the separating character is a comma (,).

To import data, you must tell the system how the input file is formatted. This is done by defining a *schema*. Schemas are constructed as a list of tuples, each of which contains pairs of *ASCII*-character names and data types (see *Valid Data Types*), ordered according to the order of columns in the input file.

Given a file *datasets/small_songs.csv* whose contents look like this:

```
1, "Easy on My Mind"
2, "No Rest For The Wicked"
3, "Does Your Chewing Gum"
4, "Gypsies, Tramps, and Thieves"
5, "Symphony No. 5"
```

Create a variable to hold the file name (for easier reuse):

```
>>> my_data_file = "datasets/small_songs.csv"
```

Create the schema *my_schema* with two columns: *id* (int32), and *title* (str):

```
>>> my_schema = [('id', ta.int32), ('title', str)]
```

The schema and file name are used in the *CsvFile()* command to describe the file format:

```
>>> my_csv_description = ta.CsvFile(my_data_file, my_schema)
```

The data fields are separated by a character delimiter. The default delimiter to separate column data is a comma. It can be changed with the parameter *delimiter*:

```
>>> my_csv_description = ta.CsvFile(my_data_file, my_schema, delimiter = ",")
```

This can be helpful if the delimiter is something other than a comma, for example, `\t` for tab-delimited records.

Occasionally, there are header lines in the data file. For example, these lines may describe the source or format of the data. If there are lines at the beginning of the file, they should be skipped by the import mechanism. The number of lines to skip is specified by the *skip_header_lines* parameter:

```
>>> csv_description = ta.CsvFile(my_data_file, my_schema, skip_header_lines = 5)
```

Now we use the schema and the file name to create *CsvFile* classes, which define the data layouts:

```
>>> my_csv = ta.CsvFile(my_data_file, my_schema)
>>> csv1 = ta.CsvFile(file_name="data1.csv", schema=schema_ab)
>>> csv2 = ta.CsvFile(file_name="more_data.txt", schema=schema_ab)

>>> raw_csv_data_file = "datasets/my_data.csv"
>>> column_schema_list = [("x", ta.float64), ("y", ta.float64), ("z", str)]
>>> csv4 = ta.CsvFile(file_name=raw_csv_data_file,
... schema=column_schema_list, delimiter='/', skip_header_lines=2)
```

8.3 Frames

A *Frame* (*capital F*) is a class of objects capable of accessing and controlling a *frame* (*lower case f*) containing “big data”. The frame is visualized as a two-dimensional table structure of rows and columns. Trusted Analytics can handle frames with large volumes of data, because it is designed to work with data spread over multiple machines.

8.3.1 Create A Frame

There are several ways to create frames:

1. as “empty”, with no schema or data
2. with a schema and data
3. by copying (all or a part of) another frame
4. as a return value from a Frame-based method; this is part of the ETL data flow.

See the Python API [Frames section](#) for more information.

Examples:

Create an empty frame:

```
>>> my_frame = ta.Frame()
```

The Frame *my_frame* is now a Python object which references an empty frame that has been created on the server.

For an example, to create a frame defined by the schema *my_csv* (see [Importing a CSV file.](#)), import the data, give the frame the name *myframe*, and create a Frame object, *my_frame*, to access it:

```
>>> my_frame = ta.Frame(source=my_csv, name='myframe')
```

To copy the frame *myframe*, create a Frame *my_frame2* to access it, and give it a new name, because the name must always be unique:

```
>>> my_frame2 = my_frame.copy(name = "copy_of_myframe")
```

To create a new frame with only columns *x* and *z* from the original frame *myframe*, and save the Frame object as *my_frame3*:

```
>>> my_frame3 = my_frame.copy(['x', 'z'], name = "copy2_of_myframe")
```

To create a frame copy of the original columns *x* and *z*, but only those rows where *z* is *TRUE*:

```
>>> my_frame4 = my_frame.copy(['x', 'z'], where = (lambda row: "TRUE" in row.z),  
... name = "copy_of_myframe_true")
```

Frames (capital ‘F’) are not the same thing as frames (lower case ‘f’). Frames (lower case ‘f’) contain data, viewed similarly to a table, while Frames are descriptive pointers to the data. Commands such as *f4 = my_frame* will only give you a copy of the Frame proxy pointing to the same data.

Let’s create a Frame and check it out:

```
>>> small_songs = ta.Frame(my_csv, name = "small_songs")  
>>> small_songs.inspect()  
>>> small_songs.get_error_frame().inspect()
```

Append:

The [append](#) method adds rows and columns of data to a frame. Columns and rows are added to the database structure, and data is imported as appropriate. If columns are the same in both name and data type, the appended data will go into the existing column.

As an example, let’s start with a frame containing two columns *a* and *b*. The frame can be accessed by Frame *my_frame1*. We can look at the data and structure of the database by using the [inspect](#) method:

```
>>> my_frame1.inspect()

a:str    b:ta.int64
/-----/
apple      182
bear       71
car       2048
```

Given another frame, accessed by Frame *my_frame2* with one column *c*:

```
>>> my_frame2.inspect()

c:str
/-----/
dog
cat
```

With *append*:

```
>>> my_frame1.append(my_frame2)
```

The result is that the first frame would have the data from both frames. It would still be accessed by Frame *my_frame1*:

```
>>> my_frame1.inspect()

a:str    b:ta.int64    c:str
/-----/
apple      182         None
bear       71         None
car       2048         None
None       None        dog
None       None        cat
```

Try this example with data files *objects1.csv* and *objects2.csv*:

```
>>> objects1 = ta.Frame(ta.CsvFile("datasets/objects1.csv",
... schema=[('Object', str), ('Count', ta.int64)],
... skip_header_lines=1), 'objects1')
>>> objects2 = ta.Frame(ta.CsvFile("datasets/objects2.csv",
... schema=[('Thing', str)], skip_header_lines=1), 'objects2')

>>> objects1.inspect()
>>> objects2.inspect()

>>> objects1.append(objects2)
>>> objects1.inspect()
```

See also the *join* method in the [API](#) section.

8.3.2 Inspect The Data

Trusted Analytics provides several methods that allow you to inspect your data, including *inspect* and *take*. The Frame class also contains frame information like *row_count*.

Examples

To see the number of rows:

```
>>> objects1.row_count
```

To see the number of columns:

```
>>> len(objects1.schema)
```

To see all the Frame data:

```
>>> objects1
```

To see two rows of data:

```
>>> objects1.inspect(2)
```

Gives you something like this:

```
a:ta.float64  b:ta.int64
/-----/
      12.3000          500
      195.1230        183954
```

Using the `take()` method, makes a list of lists of frame data. Each list has the data from a row in the frame accessed by the Frame, in this case, 3 rows beginning from row 2.

```
>>> subset_of_objects1 = objects1.take(3, offset=2)
>>> print subset_of_objects1
```

Gives you something like this:

```
[[12.3, 500], [195.123, 183954], [12.3, 500]]
```

Note: The row sequence of the data is NOT guaranteed to match the sequence of the input file. When the data is spread out over multiple clusters, the original sequence of rows from the raw data is lost. Also, the sequence order of the columns is changed (from original data) by some commands.

Some more examples to try:

```
>>> animals = ta.Frame(ta.CsvFile("datasets/animals.csv",
... schema=[('User', ta.int32), ('animals', str), ('int1', ta.int64),
... ('int2', ta.int64), ('Float1', ta.float64), ('Float2',
... ta.float64)], skip_header_lines=1), 'animals')
>>> animals.inspect()
>>> freq = animals.top_k('animals', animals.row_count)
>>> freq.inspect(freq.row_count)

>>> from pprint import *
>>> summary = {}
>>> for col in ['int1', 'int2', 'Float1', 'Float2']:
...     summary[col] = animals.column_summary_statistics(col)
...     pprint(summary[col])
```

8.3.3 Clean The Data

The process of “data cleaning” encompasses the identification and removal or repair of incomplete, incorrect, or malformed information in a data set. The Trusted Analytics Python API provides much of the functionality necessary for these tasks. It is important to keep in mind that it was designed for data scalability. Thus, using external Python packages for these tasks, while possible, may not provide the same level of efficiency.

Warning: Unless stated otherwise, cleaning functions use the Frame proxy to operate directly on the data, so they change the data in the frame, rather than return a new frame with the changed data. It is recommended that you copy the data to a new frame on a regular basis and work on the new frame. This way, you have a fall-back if something does not work as expected:

```
>>> next_frame = current_frame.copy()
```

In general, the following functions select rows of data based upon the data in the row. For details about row selection based upon its data see [Python User Functions](#).

Example of data cleaning:

```
>>> def clean_animals(row):
...     if 'basset hound' in row.animals:
...         return 'dog'
...     elif 'guinea pig' in row.animals:
...         return 'cavy'
...     else:
...         return row.animals

>>> animals.add_columns(clean_animals, ('animals_cleaned', str))
>>> animals.drop_columns('animals')
>>> animals.rename_columns({'animals_cleaned' : 'animals'})
```

Drop Rows:

The *drop_rows* method takes a predicate function and removes all rows for which the predicate evaluates to True.

Examples:

Drop any rows in the animals frame where the value in column *int2* is negative:

```
>>> animals.drop_rows(lambda row: row['int2'] < 0)
```

To drop any rows where *a* is empty:

```
>>> my_frame.drop_rows(lambda row: row['a'] is None)
```

To drop any rows where any column is empty:

```
>>> my_frame.drop_rows(lambda row: any([cell is None for cell in row]))
```

Filter Rows:

The *filter* method is like *drop_rows*, except it removes all rows for which the predicate evaluates to False.

Examples:

To delete those rows where field *b* is outside the range of 0 to 10:

```
>>> my_frame.filter(lambda row: 0 >= row['b'] >= 10)
```

Drop Duplicates:

The `drop_duplicates` method performs a row uniqueness comparison across the whole table.

Examples:

To drop any rows where the data in *a* and column *b* are duplicates of some previously evaluated row:

```
>>> my_frame.drop_duplicates(['a', 'b'])
```

To drop all duplicate rows where the columns *User* and *animals* are duplicate:

```
>>> animals.drop_duplicates(['User', 'animals'])
>>> animals.inspect(animals.row_count)
```

Drop Columns:

Columns can be dropped either with a string matching the column name or a list of strings:

```
>>> my_frame.drop_columns('b')
>>> my_frame.drop_columns(['a', 'c'])
```

Rename Columns:

Columns can be renamed by giving the existing column name and the new name, in the form of a dictionary. Unicode characters should not be used for column names.

Rename *a* to “id”:

```
>>> my_frame.rename_columns({'a': 'id'})
```

Rename column *b* to “author” and *c* to “publisher”:

```
>>> my_frame.rename_columns({'b': 'author', 'c': 'publisher'})
```

8.3.4 Transform The Data

Often, you will need to create new data based upon the existing data. For example, you need the first name combined with the last name, or you need the number of times John spent more than five dollars, or you need the average age of students attending a college.

Add Columns:

Columns can be added to the frame using values from other columns as their value.

Add a column *int1_times_int2* as an `ta.float64` and fill it with the contents of column *int1* and column *int2* multiplied together:

```
>>> animals.add_columns(lambda row: row.int1*row.int2, ('int1xint2',
... ta.float64))
```

Add a new column *all_ones* and fill the entire column with the value 1:

```
>>> animals.add_columns(lambda row: 1, ('all_ones', ta.int64))
```

Add a new column *float1_plus_float2* and fill the entire column with the value of column *float1* plus column *float2*, then save a summary of the frame statistics:

```
>>> animals.add_columns(lambda row: row.Float1 + row.Float2,
... ('Float1PlusFloat2', ta.float64))
>>> summary['Float1PlusFloat2'] =
... animals.column_summary_statistics('Float1PlusFloat2')
```

Add a new column *pwl*, type *ta.float64*, and fill the value according to this table:

value in column <i>float1_plus_float2</i>	value for column <i>pwl</i>
None	None
Less than 50	<i>float1_plus_float2</i> times 0.0046 plus 0.4168
At least 50 and less than 81	<i>float1_plus_float2</i> times 0.0071 plus 0.3429
At least 81	<i>float1_plus_float2</i> times 0.0032 plus 0.4025
None of the above	None

An example of Piecewise Linear Transformation:

```
>>> def piecewise_linear_transformation(row):
...     x = row.float1_plus_float2
...     if x is None:
...         return None
...     elif x < 50:
...         m, c = 0.0046, 0.4168
...     elif 50 <= x < 81:
...         m, c = 0.0071, 0.3429
...     elif 81 <= x:
...         m, c = 0.0032, 0.4025
...     else:
...         return None
...     return m * x + c

>>> animals.add_columns(piecewise_linear_transformation, ('pwl', ta.float64))
```

Create multiple columns at once by making a function return a list of values for the new frame columns:

```
>>> animals.add_columns(lambda row: [abs(row.int1), abs(row.int2)],
... [('abs_int1', ta.int64), ('abs_int2', ta.int64)])
```

8.3.5 Examining the Data

To get standard descriptive statistics information about *my_frame*, use the frame function *column_summary_statistics*:

```
>>> my_frame.column_summary_statistics()
```

Group by (and aggregate):

Rows can be grouped together based on matching column values, after which an aggregation function can be applied on each group, producing a new frame.

Example process of using aggregation based on columns:

1. given our frame of animals

2. create a new frame and a Frame *grouped_animals* to access it
3. group by unique values in column *animals*
4. average the grouped values in column *int1* and save it in a column *int1_avg*
5. add up the grouped values in column *int1* and save it in a column *int1_sum*
6. get the standard deviation of the grouped values in column *int1* and save it in a column *int1_stdev*
7. average the grouped values in column *int2* and save it in a column *int2_avg*
8. add up the grouped values in column *int2* and save it in a column *int2_sum*

```
>>> grouped_animals = animals.groupby('animals', {'int1': [ta.agg.avg,
... ta.agg.sum, ta.agg.stdev], 'int2': [ta.agg.avg, ta.agg.sum]})
>>> grouped_animals.inspect()
```

Note: The only columns in the new frame will be the grouping column and the generated columns. In this case, regardless of the original frame size, you will get six columns.

Example process of using aggregation based on both column and row together:

1. Using our data accessed by *animals*, create a new frame and a Frame *grouped_animals2* to access it
2. Group by unique values in columns *animals* and *int1*
3. Using the data in the *float1* column, calculate each group's average, standard deviation, variance, minimum, and maximum
4. Count the number of rows in each group and put that value in column *int2_count*
5. Count the number of distinct values in column *int2* for each group and put that number in column *int2_count_distinct*

```
>>> grouped_animals2 = animals.groupby(['animals', 'int1'], {'Float1':
... [ta.agg.avg, ta.agg.stdev, ta.agg.var, ta.agg.min, ta.agg.max],
... 'int2': [ta.agg.count, ta.agg.count_distinct]})
```

Example process of using aggregation based on row:

1. Using our data accessed by *animals*, create a new frame and a Frame *grouped_animals2* to access it
2. Group by unique values in columns *animals* and *int1*
3. Count the number of rows in each group and put that value in column *count*

```
>>> grouped_animals2 = animals.groupby(['animals', 'int1'],
... ta.agg.count)
```

Note: `agg.count` is the only full row aggregation function supported at this time.

Aggregation currently supports using the following functions:

- avg
- count
- count_distinct
- max
- min
- stdev
- sum
- var (see glossary *Bias vs Variance*)

Join:

Create a **new** frame from a *join* operation with another frame.

Given two frames *my_frame* (columns *a*, *b*, *c*) and *your_frame* (columns *b*, *c*, *d*). For the sake of readability, in these examples we will refer to the frames and the Frames by the same name, unless needed for clarity:

```
>>> my_frame.inspect()

a:str      b:str      c:str
/-----/
alligator  bear       cat
auto       bus        car
apple      berry     cantaloupe
mirror     frog       ball

>>> your_frame.inspect()

b:str      c:ta.int64    d:str
/-----/
bus        871      dog
berry     5218    frog
blue      0       log
```

Column *b* in both frames is a unique identifier used to relate the two frames. Following this instruction will join *your_frame* to *my_frame*, creating a new frame with a new Frame to access it, with all of the data from *my_frame* and only that data from *your_frame* which has a value in *b* that matches a value in *my_frame b*:

```
>>> our_frame = my_frame.join(your_frame, 'b', how='left')
```

The result is *our_frame*:

```
>>> our_frame.inspect()

a:str      b:str      c_L:str      c_R:ta.int64    d:str
/-----/
alligator  bear       cat          None           None
auto       bus        car          871            dog
apple      berry     cantaloupe   5281           frog
mirror     frog       ball         None           None
```

Doing an “inner” join this time will include only data from *my_frame* and *your_frame* which have matching values in *b*:

```
>>> inner_frame = my_frame.join(your_frame, 'b')
```

or

```
>>> inner_frame = my_frame.join(your_frame, 'b', how='inner')
```

Result is *inner_frame*:

```
>>> inner_frame.inspect()

a:str      b:str      c_L:str      c_R:ta.int64    d:str
/-----/
auto       bus        car          871            dog
apple      berry     cantaloupe   5218           frog
```

Doing an “outer” join this time will include only data from *my_frame* and *your_frame* which do not have matching values in *b*:

```
>>> outer_frame = my_frame.join(your_frame, 'b', how='outer')
```

Result is *outer_frame*:

```
>>> outer_frame.inspect()

a:str      b:str      c_L:str      c_R:ta.int64  d:str
/-----/
alligator  bear      cat          None          None
mirror     frog      ball         None          None
None       blue     None         0             log
```

If column *b* in *my_frame* and column *d* in *your_frame* are the common column: Doing it again but including all data from *your_frame* and only that data in *my_frame* which has a value in *b* that matches a value in *your_frame d*:

```
>>> right_frame = my_frame.join(your_frame, left_on='b', right_on='d',
... how='right')
```

Result is *right_frame*:

```
>>> right_frame.inspect()

a:str      b_L:str      c:str      b_R:str      c:ta.int64  d:str
/-----/
None       None         None       bus          871         dog
mirror     frog         ball       berry        5218        frog
None       None         None       blue         0           log
```

Flatten Column:

The function `flatten_column` creates a **new** frame by splitting a particular column and returns a Frame object. The column is searched for rows where there is more than one value, separated by commas. The row is duplicated and that column is spread across the existing and new rows.

Given a frame accessed by Frame *my_frame* and the frame has two columns *a* and *b*. The “original_data”:

```
1-"solo,mono,single"
2-"duo,double"
```

Bring the data in where it can by worked on:

```
>>> my_csv = ta.CsvFile("original_data.csv", schema=[('a', ta.int64),
... ('b', str)], delimiter='-')
>>> my_frame = ta.Frame(source=my_csv)
```

Check the data:

```
>>> my_frame.inspect()

a:ta.int64  b:string
/-----/
1          solo, mono, single
2          duo, double
```

Spread out those sub-strings in column *b*:

```
>>> your_frame = my_frame.flatten_column('b')
```

Now check again and the result is:

```
>>> your_frame.inspect()

  a:ta.int64  b:str
/-----/
1          solo
1          mono
1          single
2          duo
2          double
```

8.4 Seamless Graph

For the examples below, we will use a Frame *my_frame*, which accesses an arbitrary frame of data consisting of the following:

Employee	Manager	Title	Years
Bob	Steve	Associate	1
Jane	Steve	Sn Associate	3
Anup	Steve	Associate	3
Sue	Steve	Market Analyst	1
Mohit	Steve	Associate	2
Steve	David	Marketing Manager	5
Larry	David	Product Manager	3
David	Rob	VP of Sales	7

8.4.1 Build the Graph

Make an empty graph and give it a name:

```
>>> my_graph = ta.graph()
>>> my_graph.name = "personnel"
```

Define the vertex types:

```
>>> my_graph.define_vertex_type("employee")
>>> my_graph.define_vertex_type("manager")
>>> my_graph.define_vertex_type("title")
>>> my_graph.define_vertex_type("years")
```

Define the edge type:

```
>>> my_graph.define_edge_type('worksunder', 'Employee', 'Employee', directed=True)
```

Add the data:

```
>>> my_graph.vertices['Employee'].add_vertices(employees_frame,
... 'Employee', ['Title'])
>>> my_graph.edges['worksunder'].add_edges(employees_frame, 'Employee',
... 'Manager', ['Years'], create_missing_vertices = True)
```

Warning: Improperly built graphs can give inconsistent results. For example, given EdgeFrames with this data:

```
Movieid, movieTitle, Rating, userId
1, Titanic, 3, 1
1, My Own Private Idaho, 3, 2
```

If the vertices are built out of this data, the vertex with Movieid of 1 would sometimes have the Titanic data and sometimes would have the Idaho data, based upon which order the records are delivered to the function.

8.4.2 Other Graph Options

Inspect the graph:

```
>>> my_graph.vertex_count
>>> my_graph.edge_count
>>> my_graph.vertices['Employee'].inspect(20)
>>> my_graph.edges['worksunder'].inspect(20)
```

For further information, see the API section on [Graphs](#). Export the graph to a TitanGraph:

```
>>> my_titan_graph = my_graph.export_to_titan("titan_graph")
```

Make a VertexFrame:

```
>>> my_vertex_frame = my_graph.vertices("employee")
```

Make an EdgeFrame:

```
>>> my_edge_frame = my_graph.edges("worksunder")
```

8.5 Titan Graph

For the examples below, we will use a Frame *my_frame*, which accesses an arbitrary frame of data consisting of the following:

Employee	Manager	Title	Years
Bob	Steve	Associate	1
Jane	Steve	Sn Associate	3
Anup	Steve	Associate	3
Sue	Steve	Market Analyst	1
Mohit	Steve	Associate	2
Steve	David	Marketing Manager	5
Larry	David	Product Manager	3
David	Rob	VP of Sales	7

8.5.1 Graph Creation

A Titan graph is created by exporting it from a seamless graph. For further information, as well as Titan graph attributes and methods, see the API section on [Titan Graph](#).

8.6 Graph Analytics

See these API sections for individual analytic processes:

- *Clustering Coefficients*
- *Connected Components (CC)*
- *Degree Calculation*
- *K-Clique Percolation*
- *PageRank (PR)*

8.6.1 Python User Functions

Table of Contents

- *Frame Row UDF*
 - *Row Object Parameter*
- *UDF Guidelines*

A *UDF* is a Python function written by the user on the client-side which can execute in a distributed fashion on the cluster. The function is serialized and copies are distributed throughout the cluster as part of command execution. Various API command methods accept a UDF as a parameter. A UDF runs under the constraints of the particular command.

Frame Row UDF

A Frame Row UDF is a UDF which operates on a single row of a frame. The function has one parameter, a *row* object. Here is an example of a Row UDF that returns True for a row where the column named “score” has a value greater than zero:

```
>>> def my_custom_row_func(row):
...     return row['score'] > 0
```

This function would be useful in a Frame filter command, which filters a data frame keeping only those rows which meet certain criteria, – in this case, only rows with scores greater than zero:

```
>>> my_csv = CsvFile("tresults.txt", [('test', str), ('score', int32)])
>>> my_frame = Frame(my_csv)
>>> my_frame.filter(my_custom_row_func)
```

The filter command iterates over every row in the frame and evaluates the user-defined function on each one and keeps only those rows which evaluate to True.

Row Object Parameter

The Row object is a read-only dictionary-like structure which contains the cell values for a particular row. The values are accessible using the column name, with typical Python square bracket lookup, as shown in the example above. The value of cell in column ‘score’ is accessed like this:

```
>>> row['score']
```

The cell values may also be accessed using *dot-member* notation. Here is an equivalent row function:

```
>>> def my_custom_row_func2(row):  
...     return row.score > 0
```

The *dot-member* notation is provided for convenience (it follows the pandas DataFrame technique) and only works for columns whose names are legal Python variable names (it does not start with a number and is composed of alphanumeric characters and the underscore character). Columns whose names do not meet this criteria must be referenced using square brackets with strings.

New values must be added to a frame using the Frame's `add_columns` method.

The *row* object supports a few dictionary-like methods:

- `keys()` – returns a list of column names
- `values()` – returns a list of column values
- `items()` – returns a list of (key, value) tuples
- `types()` – returns a list of column types

These methods all produce lists in the same order, in other words, it is safe to correlate their indices.

Also, iterating on the row object is the equivalent of iterating on `items()`. For example:

```
>>> def row_sum(row):  
...     """  
...     sums the values in the row, except for column "name"  
...     """  
...     try:  
...         s = 0  
...         for k, v in row:  
...             if k != 'name':  
...                 sum += v  
...         return s  
...     except:  
...         return -1  
  
>>> frame.add_columns(row_sum, ('sum', int32))
```

Note: This example is for illustration only. There are other, perhaps more Pythonic, ways of doing this, like using a list comprehension.

UDF Guidelines

Here are some guidelines to follow when writing a UDF:

1. **Error handling:** Include error handling. If the function execution raises an exception, it will cause the entire command to fail and possibly leave the frame or graph in an incomplete state. The best practice is to put all UDF functionality in a `try: except: block`, where the `except: clause` returns a default value or performs a benign side effect. See the `row_sum` function example above, where we used a `try: except: block` and produced a -1 for rows which caused errors.
2. **Dependencies:** All dependencies used in the UDF must be available in **the same Python code file** as the UDF or available in the server's installed Python libraries. The serialization technique to get the code distributed throughout the cluster will only serialize dependencies in the same Python module (in other words, file) right now.

3. **Simplicity:** Stay within the intended simple context of the given command, like a row operation. Do not try to call other API methods or perform fancy system operations (which will fail due to permissions).
4. **Performance:** Be mindful of performance. These functions execute on every row of data, in other words, several times.
5. **Printing:** Printing (to stdout, stderr, ...) within the UDF will not show up in the client REPL. Such messages will usually end up in the server logs. In general, avoid printing.
6. **Lambda:** Lambda syntax is valid, but discouraged:

```
>>> frame.filter(lambda row: row.score > 0)
```

This is legal and attractively shorter to write. However, lambdas do not provide error handling, nor do they have a “name” that would be useful in exception stack traces. They cannot be tested in isolation nor have embedded documentation. Lambdas are not very shareable.

7. **Closures:** Closures are read-only. Any closed over variables are copied during serialization, so it is not possible to obtain side-effects.
8. **Multiple executions:** Do not make any assumptions about how many times the function may get executed.
9. **Parameterizing a UDF:** Parameterizing a UDF is possible using Python techniques of closures and nesting function definitions. For example, the Row UDF only takes a single row object parameter. It could be useful to have a row function that takes a few other parameters. Let’s augment the row_sum function above to take a list of columns to ignore:

```
>>> def get_row_sum_func(ignore_list):
...     """
...     returns a row function which sums the values in the row,
...     except for ignored columns
...     """
...     def row_sum2(row):
...         try:
...             s = 0
...             for k, v in row:
...                 if k not in ignore_list:
...                     s += v
...             return s
...         except:
...             return -1
...     return row_sum2

>>> frame.add_columns(get_row_sum_func(['name', 'address']), ('sum', int32))
```

The row_sum2 function closes over the *ignore_list* argument making it available to the row function that executes on each row.

MACHINE LEARNING

Machine learning is the study of constructing algorithms that can learn from data.

When someone uses a search engine to perform a query, they are returned a ranked list of websites, ordered according to predicted relevance. Ranking these sites is typically done using page content, as well as the relevance of other sites that link to a particular page. Machine learning is used to automate this process, allowing search engine companies to scale this process up to billions of potential web pages.

Online retailers often use a machine learning algorithm called collaborative filtering to suggest products users might be interested in purchasing. These suggestions are produced dynamically and without the use of a specific input query, so retailers use a customer's purchase and browsing history, along with those of customers with whom shared interests can be identified. Implementations of collaborative filtering enable these recommendations to be done automatically, without directly involving analysts.

There are many other problems that are amenable to *machine learning* solutions. Translation of text for example is a difficult issue. A corpus of pre-translated text can be used to teach an algorithm a mapping from one language to another.

9.1 Algorithms

9.1.1 Machine Learning Algorithms

Table of Contents

- *Collaborative Filtering*
- *Graphical Models*
- *Topic Modeling*

Collaborative Filtering

See the models section of the API for details.

Graphical Models

Graphical models find more insights from structured noisy data. See graph API for details of the *Label Propagation* (LP) and *Loopy Belief Propagation* (LBP).

Topic Modeling

For Topic Modeling, see the LDA Model section of the API and http://en.wikipedia.org/wiki/Topic_model

9.2 Supervision

Trusted Analytics incorporates supervised, unsupervised, and semi-supervised machine learning algorithms. Supervised algorithms are used to learn the relationship between features in a dataset and some labeling schema, such as is in classification. For example, binary logistic regression builds a model for relating a linear combination of input features (e.g., high and low temperatures for a collection of days) to a known binary label (e.g., whether or not someone went for a trail run on that day). Once the relationship between temperature and running activity is learned, then the model can be used to make predictions about new running activity, given the days temperatures. Unsupervised machine learning algorithms are used to find patterns or groupings in data for which class labels are unknown. For example, given a data set of observations about flowers (e.g., petal length, petal width, sepal length, and sepal width), an unsupervised clustering algorithm could be used to cluster observations according to similarity. Then, a researcher could look for reasonable patterns in the groupings, such as “similar species appear to cluster together.” Semi-supervised learning is the natural combination of these two classes of algorithms, in which unlabeled data are supplemented with smaller amounts of labeled data, with the goal of increasing the accuracy of learning. For more information on these approaches, the respective wikipedia entries to these approaches provide an easy-to-read overview of their strengths and limitations.

9.3 Other Resources

There is plenty of literature on *machine learning* for those who want to gain a more thorough understanding of it. We recommend: [Introduction to Machine Learning](http://alex.smola.org/drafts/thebook.pdf)¹ and [Wikipedia: Machine Learning](http://en.wikipedia.org/wiki/Machine_Learning)². You might find this link helpful as well: [Everything You Wanted to Know About Machine Learning, But Were Too Afraid To Ask \(Part Two\)](http://blog.bigml.com/2013/02/21/everything-you-wanted-to-know-about-machine-learning-but-were-too-afraid-to-ask-part-two/)³.

¹<http://alex.smola.org/drafts/thebook.pdf>

²http://en.wikipedia.org/wiki/Machine_learning

³<http://blog.bigml.com/2013/02/21/everything-you-wanted-to-know-about-machine-learning-but-were-too-afraid-to-ask-part-two/>

BEST KNOWN METHODS (USER)

Table of Contents

- *Python*
 - *Server Connection*
 - *Errors*
 - *Tab Completion*

10.1 Python

10.1.1 Server Connection

Ping the server:

```
>>> import trustedanalytics as ta
>>> ta.server.ping()
Successful ping to Trusted Analytics ATK at http://localhost:9099/info
>>> ta.connect()
```

View and edit the server connection:

```
>>> print ta.server
host:    localhost
port:    9099
scheme:  http
version: v1

>>> ta.server.host
'localhost'

>>> ta.server.host = '10.54.99.99'
>>> ta.server.port = None
>>> print ta.server
host:    10.54.99.99
port:    None
scheme:  http
version: v1
```

Reset configuration back to defaults:

```
>>> ta.server.reset()
>>> print ta.server
host:    localhost
port:    9099
scheme:  http
version: v1
```

10.1.2 Errors

By default, the toolkit does not print the full stack trace when exceptions occur. To see the full Python stack trace of the last (i.e. most recent) exception:

```
>>> ta.errors.last
```

To enable always printing the full Python stack trace, set the *show_details* property:

```
>>> import trustedanalytics as ta

# show full stack traces
>>> ta.errors.show_details = True

>>> ta.connect()

# ... the rest of your script ...
```

If you enable this setting at the top of your script you get better error messages. The longer error messages are really helpful in bug reports, emails about issues, etc.

10.1.3 Tab Completion

Allows you to use the tab key to complete your typing for you.

If you are running with a standard Python REPL (not iPython, bPython, or the like) you will have to set up the tab completion manually:

Create a `.pythonrc` file in your home directory with the following contents:

```
>>> import rlcompleter, readline
>>> readline.parse_and_bind('tab:complete')
```

Or you can just run the two lines in your REPL session.

This will let you do the tab completion, but will also remember your history over multiple sessions:

```
# Add auto-completion and a stored history file of commands to your Python
# interactive interpreter. Requires Python 2.0+, readline.

>>> import atexit
>>> import os
>>> import readline
>>> import rlcompleter
>>> import sys

# Autocomplete is bound to the Esc key by default, so change it to tab.
>>> readline.parse_and_bind("tab: complete")

>>> historyPath = os.path.expanduser("~/pyhistory")
```

```
>>> def save_history(historyPath=historyPath):  
...     import readline  
...     readline.write_history_file(historyPath)  
  
>>> if os.path.exists(historyPath):  
...     readline.read_history_file(historyPath)  
  
>>> atexit.register(save_history)  
  
# anything not deleted (sys and os) will remain in the interpreter session  
>>> del atexit, readline, rlcompleter, save_history, historyPath
```

Note: If the `.pythonrc` does not take effect, add `PYTHONSTARTUP` in your `.bashrc` file:

```
export PYTHONSTARTUP=~/.pythonrc
```


Part III

Extending Trusted Analytics

TRACKING METADATA

Table of Contents

- *Using PostgreSQL*
- *Basic PSQL commands*

By default, Trusted Analytics uses H2, an in-memory database that is lost on application restart. This is convenient for testing. H2 setup is completely automatic. No steps below are needed for H2.

The configuration script, used during normal installation (see *REST Server Configuration*), changes this behavior to use PostgreSQL.

11.1 Using PostgreSQL

By default, the Cloudera Manager installs [PostgreSQL](http://www.postgresql.org)¹ which is used for tracking metadata. PostgreSQL is only required if your engine and API server are on different nodes or if you are in a more production environment (where you want your data to persist between restarts).

- On your ‘./interfaces/src/main/resources/reference.conf’ or your ‘application.conf’ (if you are using RPM packages) set the following:

```
metastore.connection-postgresql.host = "localhost"  
metastore.connection.url = "jdbc:postgresql://"${trustedanalytics.atk.metastore.  
    connection-postgresql.host}":"${trustedanalytics.atk.metastore.connection-  
    postgresql.port}"/"${trustedanalytics.atk.metastore.connection-postgresql.  
    database}
```

The `metastore.connection.url` line above was broken across multiple lines to assist display on various media. The line should be entered as a single line with no gaps (spaces).

- Configure PostgreSQL to use password authentication
 - Verify PostgreSQL is already installed:

```
yum info postgresql
```

If PostgreSQL is not already installed, you will need to install it before going on.

- If the data folder ‘`cd /var/lib/pgsql/data`’ is empty, you need to run:

```
sudo service postgresql initdb
```

¹<http://www.postgresql.org>

- Switch to the Postgres user:

```
sudo su postgres
```

- cd to the data folder:

```
cd /var/lib/pgsql/data
```

- Modify *pg_hba.conf*, adding a line with the IP that PostgreSQL will listen on for connections, add this **before** the other lines in the file.

- * If Engine and PostgreSQL are on the same node:

TYPE	DATABASE	USER	CIDR-ADDRESS	METHOD
host	all	metastore	127.0.0.1/32	md5

- * If Engine and PostgreSQL are on different nodes:

TYPE	DATABASE	USER	CIDR-ADDRESS	METHOD
host	all	metastore	<IP of Engine Server>/32	md5

- Modify *postgresql.conf* and uncomment the *listen_addresses* setting with the IP that PostgreSQL should listen on.

- * If Engine and PostgreSQL are on the same node, lock PostgreSQL to listen on the local loopback interface only:

```
listen_addresses = 'localhost'
```

- * If Engine and PostgreSQL are on different nodes allow PostgreSQL to listen to both an externally accessible interface and the local loopback interface:

```
listen_addresses = 'localhost,<IP of Accessible Interface>'
```

- * or to listen on all interfaces:

```
listen_addresses = '*'
```

- Restart PostgreSQL

```
sudo service postgresql restart
```

- Create a metastore user and database

- Run *psql*
 - create user metastore with createdb with encrypted password 'atkuser123' (you can change it later)
 - create database metastore with owner metastore
 - It is also good to create a user for yourself so you don't have to `sudo` all of the time

- * create user yourUserName with superuser; // etc

- Start our application, it will create the schema automatically using Flyway (which is installed by default).

- Use *d* to see the schema, see the *cheatsheet* <ad_psql_cs>

- Insert a user:

```
psql metastore
insert into users (username, API_key, created_on, modified_on)
values( 'metastore', 'test_API_key_1', now(), now() )
```

11.2 Basic PSQL commands

PSQL has a really nice command-line interface with full help and documentation build-in.

psql dbname - launch the psql prompt and connect to a database

h - gives help on SQL, lists all of the possible commands

h create user - gives help on “create user”

? - gives help on PSQL command line

d - lists all of the objects in your database

d tableName - describes a table

dt - lists all tables

q - quit

c dbname - connect to another DB

IPYTHON SETUP INSTRUCTIONS

Table Of Contents

- *Introduction*
- *Requirements*
 - *Installing IPython*
- *Configure Trusted Analytics Python REST Client*
- *Running IPython*
 - *IPython Notebook*

12.1 Introduction

These instructions show how to use the Trusted Analytics server through [IPython](http://ipython.org/)¹. This is a guide through the IPython setup needed to communicate with an Trusted Analytics service on a remote cluster. With an understanding of this information, the Python REST client can be accessed from a remote host through an IPython shell or notebook server.

12.2 Requirements

A working Trusted Analytics cluster installation is required. It must be configured to run with the Python2.7 executable. If Trusted Analytics is not installed, see [Package Installation](#) to install it.

12.2.1 Installing IPython

To install IPython run:

```
$ sudo yum install python27-ipython
```

12.3 Configure Trusted Analytics Python REST Client

Before IPython can operate properly, it is necessary to configure the Trusted Analytics REST client. The REST client needs to know where to find the Trusted Analytics REST server. This is done by updating the host address in `‘usr/lib/trustedanalytics/rest-client/python/rest/config.py’`:

¹<http://ipython.org/>

```
$ sudo vim /usr/lib/trustedanalytics/rest-client/python/rest/config.py
```

The 'config.py' file will look similar to this:

```
"""
config file for REST client
"""
# default connection config
class server:
    host = "localhost"
    port = 9099
    scheme = "http"
    version = "v1"
    headers = {'Content-type': 'application/json',
               'Accept': 'application/json,text/plain',
               'Authorization': "test_api_key_1"}

class polling:
    start_interval_secs = 1
    max_interval_secs = 20
    backoff_factor = 1.02

build_id = None
```

Update the address for host to the [Fully Qualified Domain Name](#)² or the IP address of the node hosting the Trusted Analytics REST server.

12.4 Running IPython

Test the Trusted Analytics IPython installation by importing the REST client libraries inside of a notebook or IPython shell and ping the REST server.

```
# testing IPython/Trusted Analytics ATK

$ ipython
Python 2.7.5 (default, Sep  4 2014, 17:06:50)
Type "copyright", "credits" or "license" for more information.
IPython 2.2.0 -- An enhanced Interactive Python.
?      -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help    -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.

In [1]: import trustedanalytics as ta

In [2]: ta.server.ping()
Successful ping to Trusted Analytics ATK at http://localhost:9099/info

In [3]: ta.connect()
```

12.4.1 IPython Notebook

All the dependencies to run the IPython notebook server are also installed which lets the IPython shell be run from a web browser. The notebook server is accessed by:

²http://en.wikipedia.org/wiki/Fully_qualified_domain_name

```
$ ipython notebook
```


PLUGIN AUTHORIZING GUIDE

Table of Contents

- *Introduction*
- *Types of Plugins*
 - *Commands and Queries*
- *When to Write a Plugin*
- *Plugin Support Services*
 - *Plugin Life Cycle*
 - *Logging and Error Handling*
 - *Defaulting Arguments*
 - *Execution Flow*
 - *Accessing Spark or Other Components*
- *Creating a CommandPlugin*
 - *Naming*
 - *REST Input and Output*
 - *Frame and Graph References*
 - *Self Arguments*
 - *Single Value Results*
- *Creating an Archive*
- *Deployment*
- *Configuration*
- *Archive Declaration*
- *Enabling the Archive*

13.1 Introduction

Trusted Analytics provides an extensibility framework that allows new commands and algorithms to be added to the system at runtime, without requiring Trusted Analytics source code, nor recompiling the application.

Plug-ins should be easy to write, and should not require the author to have a deep understanding of the REST server, the execution engine, or any of its supporting libraries. In particular, plug-in authors should not have to understand issues like how to manage multiple threads of execution, user authentication, or marshaling of data to and from JSON.

Plug-ins should also be isolated from the application as a whole, as well as from other plug-ins. Each plug-in should be allowed to use whatever libraries it needs, without concern for conflicts with the libraries that Trusted Analytics uses for its own needs.

13.2 Types of Plugins

13.2.1 Commands and Queries

The most common kinds of plugins are primarily divided into two categories, commands and queries. Commands are actions that are typically initiated by user, that have some impact on the system, such as loading a data frame, or removing columns from one.

Queries are also initiated by users, but their purpose is to return data to a client, with no side effects.

The interfaces that command and query plug-ins implement are very similar, but it is important to use the correct interface so the system can preserve the expected performance and semantics.

The outputs of commands and queries, and the processing of them, are monitored by the Trusted Analytics processing engine.

13.3 When to Write a Plugin

Many of the operations and algorithms that are desirable to express can be written in Python using the Python client. However, some kinds of operations are inconvenient to express in that format, or require better performance than Python can provide.

Anytime there is a new function such as an analytical or *machine learning* algorithm that would be desirable to publish for use via the Python client or REST server, it is worth considering writing a CommandPlugin.

13.4 Plugin Support Services

13.4.1 Plugin Life Cycle

Plugins are loaded at application start up, and a `start()` method is called to perform any necessary one-time setup operations. When the application ends, a `stop()` method is called to do clean up operations. Calling `stop()`, or allowing `stop()` to complete, is not guaranteed, depending on how the server is terminated.

Each invocation resulting from a user action or other source will provide an execution context object that encapsulates the arguments passed by the user as well as other relevant metadata.

13.4.2 Logging and Error Handling

Errors that occur while running a plug-in will be trapped and reported in the same way that internal errors within Trusted Analytics are normally trapped and reported.

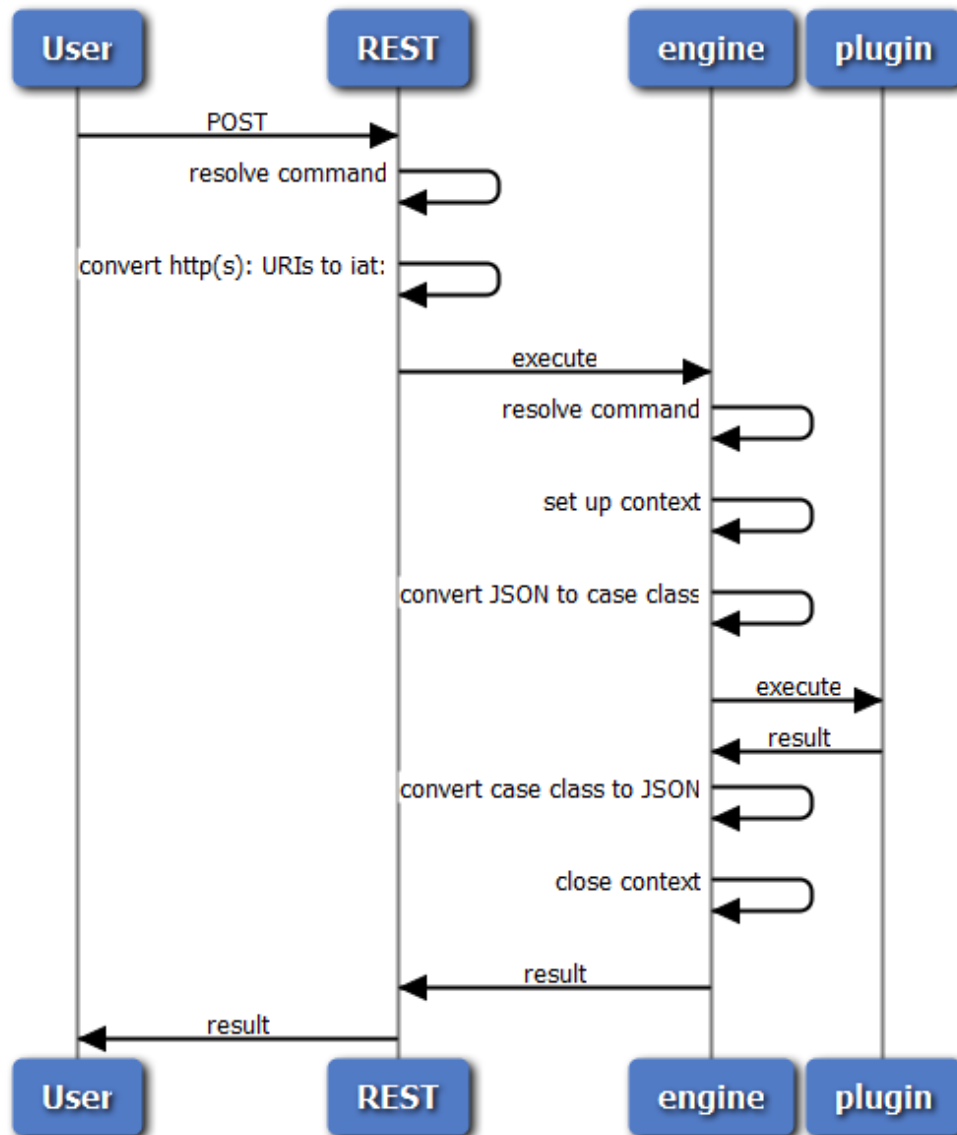
13.4.3 Defaulting Arguments

Authors should represent arguments that are not required using Option values. The system will supply default values for these optional values from the configuration system when the user's invocation does not provide them.

Configuration for commands and queries should be included in the Typesafe Config configuration file associated with the application (defaults can be provided by a `reference.conf` in the plugin's deployment jar). Configuration details are discussed in the "Configuration" section below. Plugins have access to the configuration, but only the section of it that contains settings that are relevant. For example, the Loopy Belief Propagation plugin gets its configuration from `'trustedanalytics.atk.giraph-plugins.command.graph.ml.loopy_belief_propagation.config'`. Values that appear in this

section are available to the plugin, and are passed to it during execution. The plugin does not have convenient access to other configuration parameters of the system, and plugin authors are strongly urged to take all configuration information from the Config instance they are passed rather than inspecting environment variables and so on.

13.4.4 Execution Flow



13.4.5 Accessing Spark or Other Components

For the time being, plugin authors may implement specific interfaces that declare their need for a particular service, for example, `SparkSupport` for direct access to a `SparkContext`.

See also [Best Known Methods \(Development\)](#).

13.5 Creating a CommandPlugin

13.5.1 Naming

Naming the command correctly is crucial for the usability of the system. The Python client creates Python functions to match the commands in the engine, and it places them and names them in accordance with the name specified for the plugin.

Name components are separated by slashes. For instance, the command that drops columns from a dataframe is called `dataframe/drop_column`. The Python client sees that name, knows that dataframe commands are associated with the *Frame* (*capital F*) class, and therefore generates a function named `drop_column` on the *Frame*. When the user calls that function, its arguments will be converted to JSON, sent to the REST server, and then on to the engine for processing. The results from the engine flow back through the REST server, and are converted back to Python objects.

If the name of the command contains more than one slash, the Python client will create intermediate objects that allow functions to be grouped logically together. For example, if the command is named `dataframe/ml/my_new_algorithm` (of course, real algorithms will have better names!), then the method created in the Python client could be accessed on a frame *f* using `f.ml.my_new_algorithm()`. Commands can be nested as deeply as needed, any number of intermediary objects will be created automatically so the object model of the frame or graph matches the command tree structure defined by the command names in the system.

13.5.2 REST Input and Output

Each command or query plug-in should define two case classes: one for arguments, and one for return value. The plug-in framework will ensure that the user's Python (or JSON) commands are converted into an instance of the argument class, and the output from the plug-in will also be converted back to Python (or JSON) for storage in the command execution record for later return to the client.

13.5.3 Frame and Graph References

Usually, the commands associated with a frame or graph need to accept the frame or graph on which they should operate as a parameter. Use the class `org.trustedanalytics.atk.domain.frame.FrameReference` to represent frames, and `org.trustedanalytics.atk.domain.graph.GraphReference` to represent graphs.

13.5.4 Self Arguments

Use a *FrameReference* as the type, and place this parameter first in the case class definition if it is desired that this parameter is filled by the *Frame* instance whose method is being invoked by the user. Similarly, if the method is on a graph, using a *GraphReference* in the first position will do the trick for *TitanGraph* instances.

13.5.5 Single Value Results

The result returned by command plugins can be as complex as needed. It can also be very simple — for example, a single floating point value. Since the result type of the plugin must be a case class, the convention is to return a case class with one field, which must be named “value”. When the client receives such a result, it should extract and return the single value.

13.6 Creating an Archive

Plugins are deployed in Archives – jar files that contain the plugin class, its argument and result classes, and any supporting classes it needs, along with a class that implements the Archive trait. The Archive trait provides the system with a directory of available services that the archive provides. On application start up, the application will query all the jar files it knows about (see below) to see what plugins they provide.

13.7 Deployment

Plug-Ins should be installed in the system using jar files. Jars that are found in the server’s lib directory will be available to be loaded based on configuration. The plug-ins that will be installed must be listed in the application.conf file. Each command or query advertises the location at which it would prefer to be installed in the URL structure, and if no further directives appear in configuration, they will be installed according to their request. However, using the configuration file, it is also possible to remap a plug-in to a different location or an additional location in the URL structure.

In the future, plugin discovery may be further automated, and it may also be possible to add a plugin without restarting the server.

13.8 Configuration

Server-side configuration should be stored in the reference.conf file for the plugin archive. This is a Typesafe Config file (see <https://github.com/typesafehub/config>).

13.9 Archive Declaration

Each archive should have a reference.conf file stored as a resource in its jar file. For example, in a typical Maven-based project, this file might reside in the src/main/resources folder. The Typesafe Config library automatically finds resources named “reference.conf”, so this is how the configuration file will be discovered.

The first section of the reference.conf should be the declaration of how the archive should be activated. This configuration should look like the following:

```
trustedanalytics.atk.component.archives {
  <archive-name> {
    class = "<archive-class>"
    parent = "<parent-archive>"
    config-path = "<path>"
  }
}
```

The <archive-name> is required. It should be replaced with the actual name of the archive (without the .jar suffix). For example, for graphon.jar, just use the word graphon by itself.

<archive-class> is optional. If provided, it must be the name of a class that can be found in the jar file or in its parent classloader. This class must implement the Archive trait, which makes it the archive manager. The archive manager is the service that the system uses to discover plugins in the archive. If omitted, this defaults to DefaultArchive, which uses the Config system for plugin registration and publishing.

<parent> is also optional. If provided, this archive is treated as dependent on whatever archive is specified here. For example, SparkCommand plugins should use “engine” for this entry, so that they have access to the same version of Spark the engine is using, as well as the SparkInvocation class.

<config-path> is also optional. It specifies the config path where the configuration for plugins for this archive can be found. If omitted, configuration is assumed to be included in the archive declaration block. It can be convenient to provide a value for the config path because it leads to less nested config files.

Here is a sample config file for an archive that provides a single plugin. Note that it relies on the engine archive, and re-maps its configuration to “trustedanalytics.graphon” rather than including the configuration in the trustedanalytics.atk.component.archives.graphon section.

Also note the \$-substitutions that allow configuration options from other sections to be pulled in so they’re available to the plugin.

```
trustedanalytics.atk.component.archives {
  graphon {
    parent = "engine-core"
    config-path = "trustedanalytics.graphon"
  }
}

trustedanalytics.graphon {
  command {
    available = ["graphs.sampling.vertex_sample"]
    graphs {
      sampling {
        vertex_sample {
          class = "com.trustedanalytics.spark.graphon.sampling.VertexSample"
          config {
            default-timeout = ${trustedanalytics.atk.engine.default-timeout}
            titan = ${trustedanalytics.atk.engine.titan}
          }
        }
      }
    }
  }
}

#included so that conf file can be read during unit tests,
#these will not be used when the application is actually running
trustedanalytics.atk.engine {
  default-timeout = 30s
  titan {}
}
```

13.10 Enabling the Archive

The command executor uses the config key “trustedanalytics.atk.engine.plugin.command.archives” to determine which archives it should check for command plugins. This setting is built into the reference.conf that is embedded in the engine archive (at the time of writing). For your installation, you can control this list using the application.conf file.

Once this setting has been updated, restart the server to activate the changes.

BEST KNOWN METHODS (DEVELOPMENT)

Table of Contents

- *Configuration information*

14.1 Configuration information

14.1.1 Cluster Configuration

Table of Contents

- *Installation*
 - *Requirements*
- *Help*
 - *Common Options*
 - *Help — cluster-generate*
 - *Help — cluster-config*
 - *Help — cluster-explore*
 - *Help — cluster-push*
- *Generating CDH (Cloudera Hadoop) Configurations*
 - *Script: cluster-generate*
 - * *Unique Options*
 - * *Common Options*
 - * *Example*
 - * *cdh.json*
 - *Formulas*
 - * *Cluster*
 - *Accessing Individual Configurations*
 - * *Log*
 - * *Args*
 - * *CDH*
 - * *ATK*
 - *Tips*
- *Setting CDH Configurations*
 - *Common Options*
 - * *Example*
 - *user-cdh.json*
 - * *Finding Keys*
 - * *Converting Keys*
 - * *Verifying JSON*
 - * *Create cdh.json*
 - * *Push user-cdh.json*
 - * *merged-cdh.json*
- *Exploring CDH Configurations*
 - *Common Options*

The cluster configuration tool allows easy updating and optimization of any CDH cluster from the command line. With the use of optimization formulas you can have an optimized CDH cluster in a matter of seconds.

The tool does 3 things:

- Generate optimized configurations based on your cluster hardware.
- Easy updating, restarting, and configuration deployment of CDH services.
- Simple exploration of CDH cluster configurations.

Installation

Until the pip package is hosted, you will need to clone and install directly from source.

```
git clone git@github.intel.com:gao/cluster-config.git
cd cluster-config
[sudo] python setup.py install
```


Requirements

- Python 2.7
- `argparse`¹ `>= 1.3.0`
- `cm-api`² `== 10.0.0`
- Working CDH cluster for running the script

Help

After installing, you will have three executable scripts on your path: *Generating CDH Configurations* `cluster-config`, and `cluster-explore`.

Common Options

All three scripts have mostly overlapping configurations.

- **-host**: CDH ip or host name, always required
- **-port**: CDH port, default 7180, required if your CDH manager UI port is not 7180
- **-username**: CDH username, default 'admin', required if your CDH manager UI username is not 'admin'
- **-password**: CDH manager password, if not provided on the command line you will be asked at runtime
- **-cluster**: CDH cluster name and the name of the cluster if more than one cluster is managed by cloudera manager.
- **-path**: The path we will save and load files, like `cdh.json`, default is working directory
- **-log**: Log level, defaults to INFO

Running any of the scripts with the `-help` option will provide help text for all the command line options.

Help — cluster-generate

```
$ cluster-generate --help
usage: cluster-generate [-h] [--formula FORMULA] [--formula-args FORMULA_ARGS]
                        --host HOST [--port PORT] [--username USERNAME]
                        [--password PASSWORD] [--cluster CLUSTER]
                        [--path PATH] [--log {INFO,DEBUG,WARNING,FATAL,ERROR}]

Auto generate various CDH configurations based on system resources

optional arguments:
  -h, --help                show this help message and exit
  --formula FORMULA          Auto generation formula file. Defaults to
                             /home/rodorad/Documents/config/atk-
                             config/cluster_config/formula.py
  --formula-args FORMULA_ARGS
                             Auto generation formula arguments to possibly override
                             global configurations.
  --host HOST                Cloudera Manager Host
```

¹<https://docs.python.org/2.7/library/argparse.html>

²https://github.com/cloudera/cm_api

<code>--port PORT</code>	Cloudera Manager Port
<code>--username USERNAME</code>	Cloudera Manager User Name
<code>--password PASSWORD</code>	Cloudera Manager Password
<code>--cluster CLUSTER</code>	Cloudera Manager Cluster Name if more than one cluster is managed by Cloudera Manager.
<code>--path PATH</code>	Directory where we can save/load configurations files. Defaults to working directory /home/rodorad/Documents/config/atk-config
<code>--log {INFO,DEBUG,WARNING,FATAL,ERROR}</code>	Log level [INFO DEBUG WARNING FATAL ERROR]

Help — cluster-config

```
$ cluster-config --help
usage: cluster-config [-h] --update-cdh {no,yes} --restart-cdh {no,yes}
                    [--conflict-merge {interactive,user,generated}] --host
                    HOST [--port PORT] [--username USERNAME]
                    [--password PASSWORD] [--cluster CLUSTER] [--path PATH]
                    [--log {INFO,DEBUG,WARNING,FATAL,ERROR}]

Process cl arguments to avoid prompts in automation

optional arguments:
  -h, --help                show this help message and exit
  --update-cdh {no,yes}     Should we update CDH with all configurations in
                           cdh.json/user-cdh.json?
  --restart-cdh {no,yes}    Should we restart CDH services after configuration
                           changes
  --conflict-merge {interactive,user,generated}
                           When encountering merge conflicts between the
                           generated configuration() and the user configuration()
                           what value should we default to? The 'user',
                           'generated', or 'interactive' resolution
  --host HOST               Cloudera Manager Host
  --port PORT               Cloudera Manager Port
  --username USERNAME       Cloudera Manager User Name
  --password PASSWORD       Cloudera Manager Password
  --cluster CLUSTER         Cloudera Manager Cluster Name if more than one cluster
                           is managed by Cloudera Manager.
  --path PATH               Directory where we can save/load configurations files.
                           Defaults to working directory /home/SOME_USER
  --log {INFO,DEBUG,WARNING,FATAL,ERROR}
                           Log level [INFO|DEBUG|WARNING|FATAL|ERROR]
```

Help — cluster-explore

```
$ cluster-explore --help
usage: cluster-explore [-h] [--dump DUMP] --host HOST [--port PORT]
                    [--username USERNAME] [--password PASSWORD]
                    [--cluster CLUSTER] [--path PATH]
                    [--log {INFO,DEBUG,WARNING,FATAL,ERROR}]
```

Process cl arguments to avoid prompts in automation

optional arguments:

```
-h, --help          show this help message and exit
--dump DUMP         If you want to dump all configs without asking pass
                    'yes'. Defaults to 'no'.
--host HOST         Cloudera Manager Host
--port PORT         Cloudera Manager Port
--username USERNAME Cloudera Manager User Name
--password PASSWORD Cloudera Manager Password
--cluster CLUSTER   Cloudera Manager Cluster Name if more than one cluster
                    is managed by Cloudera Manager.
--path PATH         Directory where we can save/load configurations files.
                    Defaults to working directory
                    /home/SOME_USER/
--log {INFO,DEBUG,WARNING,FATAL,ERROR}
                    Log level [INFO|DEBUG|WARNING|FATAL|ERROR]
```

Help — cluster-push

Combines cluster-generate and cluster-config

```
$ cluster-push --help
usage: cluster-push [-h] [--formula FORMULA] [--formula-args FORMULA_ARGS]
                  --update-cdh {no,yes} --restart-cdh {no,yes}
                  [--conflict-merge {interactive,user,generated}] --host
HOST [--port PORT] [--username USERNAME]
                  [--password PASSWORD] [--cluster CLUSTER] [--path PATH]
                  [--log {INFO,DEBUG,WARNING,FATAL,ERROR}]
```

Auto generate various CDH configurations based on system resources

optional arguments:

```
-h, --help          show this help message and exit
--formula FORMULA   Auto generation formula file. Defaults to
                    /home/rodorad/Documents/config/atk-
                    config/cluster_config/formula.py
--formula-args FORMULA_ARGS
                    Auto generation formula arguments to possibly override
                    global configurations.
--update-cdh {no,yes}
                    Should we update CDH with all configurations in
                    cdh.json/user-cdh.json?
--restart-cdh {no,yes}
                    Should we restart CDH services after configuration
                    changes
--conflict-merge {interactive,user,generated}
                    When encountering merge conflicts between the
                    generated configuration() and the user configuration()
                    what value should we default to? The 'user',
                    'generated', or 'interactive' resolution
--host HOST         Cloudera Manager Host
--port PORT         Cloudera Manager Port
--username USERNAME Cloudera Manager User Name
--password PASSWORD Cloudera Manager Password
--cluster CLUSTER   Cloudera Manager Cluster Name if more than one cluster
                    is managed by Cloudera Manager.
```

```
--path PATH          Directory where we can save/load configurations files.
                     Defaults to working directory
                     /home/rodorad/Documents/config/atk-config
--log {INFO,DEBUG,WARNING,FATAL,ERROR}
                     Log level [INFO|DEBUG|WARNING|FATAL|ERROR]
```

Generating CDH Configurations

Script: cluster-generate

Description: used for generating optimized CDH configurations

Unique Options

- **-formula:** path to formula file if not using the default packaged formula.
- **-formula-args:** path to formula arguments to possibly override constants.

Common Options If you forget the command line options run the script with **-help** or visit the [help section](#).

Providing only CDH connection details should be sufficient for generating optimized configurations since **-formula** is provided with a default.

The default formula can be found in the repository at *cluster_cofig/formula.py*.

Example

```
$ cluster-generate --host CLOUDERA_HOST --port CLOUDERA_PORT --username CLOUDERA_USER
What is the Cloudera manager password?
--INFO cluster selected: SOME_CLUSTER
--INFO using formula: cluster_config/formula.py
--INFO Wrote CDH configuration file to: /home/some-user/cdh.json
--INFO Wrote ATK generated configuration file to: /home/some-user/generated.conf
```

After providing the Cloudera manager password, all the cluster details will be extracted and provided to the formula to calculate an optimized configuration which gets saved to *cdh.json*.

The generated.conf file will contain ATK specific configurations. It can be ignored, if you don't intend to use the ATK server for analytics.

The optimized configurations are saved to a file to allow users to view and verify the configurations before they get saved to CDH with the *cluster-config* script.

Although we take great care to make sure the default formula will work for a majority of use cases, you should verify the settings to make sure your cluster will not be adversely affected.

Next we will read *cdh.json* and update CDH.

cdh.json Here is a chd.json file generated on a 4 node cluster

- 1 node having Cloudera manager
- 3 nodes running as workers nodes
- 32 cores
- 252 gigabytes of memory

```
{
  "HBASE": {
    "REGIONSERVER": {
      "REGIONSERVER_BASE": {
        "HBASE_REGIONSERVER_JAVA_HEAPSIZE": 26430567975
      }
    }
  },
  "YARN": {
    "GATEWAY": {
      "GATEWAY_BASE": {
        "MAPREDUCE_MAP_JAVA_OPTS_MAX_HEAP": 4129776246,
        "MAPREDUCE_MAP_MEMORY_MB": 5368709120,
        "MAPREDUCE_REDUCE_JAVA_OPTS_MAX_HEAP": 8259552492,
        "MAPREDUCE_REDUCE_MEMORY_MB": 10737418240,
        "YARN_APP_MAPREDUCE_AM_MAX_HEAP": 4129776246,
        "YARN_APP_MAPREDUCE_AM_RESOURCE_CPU_VCORES": 1,
        "YARN_APP_MAPREDUCE_AM_RESOURCE_MB": 5368709120
      }
    },
    "NODEMANAGER": {
      "NODEMANAGER_BASE": {
        "YARN_NODEMANAGER_RESOURCE_CPU_VCORES": 32,
        "YARN_NODEMANAGER_RESOURCE_MEMORY_MB": 180388626432
      }
    },
    "RESOURCEMANAGER": {
      "RESOURCEMANAGER_BASE": {
        "YARN_SCHEDULER_INCREMENT_ALLOCATION_MB": 536870912,
        "YARN_SCHEDULER_MAXIMUM_ALLOCATION_MB": 180388626432,
        "YARN_SCHEDULER_MAXIMUM_ALLOCATION_VCORES": 32,
        "YARN_SCHEDULER_MINIMUM_ALLOCATION_MB": 5368709120
      }
    }
  }
}
```

Formulas

It is possible to create your optimization formulas or copy the default formula and make your own tweaks. The formula files are regular python scripts. While the default formula is called ‘formula.py’ there are no restrictions.

The formula files are executed with `python’s exec file function`³.

Cluster The cluster variable will contain all installed service, role, config group and configuration details from the CDH cluster we are currently connected to.

Lets look at a code sample from formula.py to see how the cluster variable is being used.

```
...
hosts = cluster.yarn.nodemanager.hosts.all()
max_cores = 0
max_memory = 0
for key in hosts:
    if hosts[key].numCores > max_cores:
```

³<https://docs.python.org/2/library/functions.html#execfile>

```

    max_cores = hosts[key].numCores
    if hosts[key].totalPhysMemBytes > max_memory:
        max_memory = hosts[key].totalPhysMemBytes
...

```

On the first line with `cluster.yarn.nodemanager.hosts.all()` we are retrieving all the details for every host running the yarn node manager role. The returned object will be a list of cloudera [apihosts](#)⁴.

With the same notation you can retrieve all the details for hosts running the hbase region server role or all zookeeper servers.

- `cluster.hbase.regionserver.hosts.all()`
- `cluster.zookeeper.server.hosts.all()`

You can access all CDH services, roles, configs, and hosts with the same notation.

The pattern you want to follow is

- `cluster.SERVICE.ROLE.CONFIG_GROUP.CONFIG`
- `cluster.SERVICE.ROLE.hosts.all`
- **SERVICE** is any valid CDH [service](#)⁵
- **ROLE** is any valid CDH [role](#)⁶
- **CONFIG_GROUP** is any valid config group from [cluster-explore](#) script. A good example of a configuration group would be 'gateway_base' because just about every service has this role.
- **CONFIG** is any valid config from [cluster-explore](#) script. When accessing CDH services all attributes are lowercase while the keys displayed by the [cluster-explore](#) script are uppercase.

Accessing Individual Configurations Accessing individual configurations is easy but you must know the configuration name. You will need to run the [cluster-explore](#) script to find the full configuration name.

Here is a config key that was found with [cluster-explore](#).

```

- name: HBASE_REGIONSERVER_JAVA_HEAPSIZE
- description: Maximum size in bytes for the Java Process heap memory. Passed to Java -Xmx.
- key: HBASE.REGIONSERVER.REGIONSERVER_BASE.HBASE_REGIONSERVER_JAVA_HEAPSIZE
- value: 26430567975

```

To retrieve the following value with the cluster variable make the key all lower case and prefix cluster.

cluster.hbase.regionserver.regionserver_base.hbase_regionserver_java_heapsize.value

If you look at the implementation for the 'cluster' objects class you will notice that you can set all CDH configurations directly i would cuation against it. Saving the computed configurations to the 'cdh' dictionary will allow the user to review the configurations when they get saved to `cdh.json` and enable the saving of snapshot for administrative purposes.

Log The log variable has four methods available for sending log messages to the command line:

- `log.info`
- `log.error`

⁴http://cloudera.github.io/cm_api/apidocs/v10/ns0_apiHost.html

⁵http://cloudera.github.io/cm_api/apidocs/v10/path__clusters_-clusterName-_services.html

⁶http://cloudera.github.io/cm_api/apidocs/v10/path__clusters_-clusterName-_services_-serviceName-_roles.html

- `log.warning`
- `log.fatal`: does a `sys.exit(1)` after logging its message. All the methods take a single string as an argument.

Args `Args` is a dictionary of values provided by the user. The `args` dictionary gives us the ability to consider the users Constants when calculating configurations.

When calculating optimized configurations you often need constants to set thresholds. The default `formula.py` has several constants to allow for fine tuning.

..code:

```
NUM_THREADS = 1 # This should be set to the maximum number of multi-tenant users
OVER_COMMIT_FACTOR = 1.30
MAX_JVM_MEMORY = 32768 * MiB
MEM_FRACTION_FOR_HBASE = 0.20
MEM_FRACTION_FOR_OTHER_SERVICES = 0.20
MAPREDUCE_JOB_COUNTERS_MAX = 500
```

If the value for `'NUM_THREADS'` is given on the command line, that value is received in `'args["NUM_THREADS"]'` allowing for the decision of whether to use the default or the specified value.

CDH The `'cdh'` dictionary is the variable where the calculated CDH configurations are saved with the dictionary key being the full key to the path found with the `cluster-explore` script.

When the script has finished running, the contents of the `'cdh'` dictionary are saved to `cdh.json` so the configurations can be reviewed before updating CDH.

For example, if you wanted to save your optimized value for the hbase heap you would do this:

```
cdh[ 'HBASE.REGIONSERVER.REGIONSERVER_BASE.HBASE_REGIONSERVER_JAVA_HEAPSIZE ' ] = SOME_MEMORY_SETTING
```

Once the `execfile` has finished running, the `'cdh'` dictionary is exported to `cdh.json` which is read by the `cluster-config` or `help_cluster-push` scripts to update CDH.

ATK Much like the `'cdh'` dictionary, the `'atk'` dictionary is the place where the calculated ATK configurations are saved. If the ATK, is not being used, the dictionary can be safely ignored.

The key for every value in the `'atk'` dictionary would need to be a valid `atk configuration key`⁷.

Tips

When saving optimized configurations be aware of the expected format. For example, a memory setting for `yarn`, `hbase`, and `spark` may require different formats. `Yarn` may expect configuration values expressed in mega bytes, while `hbase` and `spark` expect values expressed in bytes.

Unfortunately, the CDH REST API does not always throw formatting errors. If there are formatting issues, login to CDH console and look for configuration warnings concerning the affected services.

Setting CDH Configurations

Script: `cluster-config`

Description: Used for updating CDH with the optimized configurations

⁷<https://github.com/intel-data/atk/blob/master/conf/examples/application.conf.tpl>

Unique Options:

- **–update-cdh**: Whether to update CDH configurations. Either yes or no.
- **–restart-cdh**: Whether to restart CDH after updating its configuration. Either yes or no.
- **–conflict-merge**: Conflict resolution preference when encountering key conflicts between `cdh.json` and `user-cdh.json`. Defaults to `user-cdh.json`. Valid values [user, generated, interactive]

Common Options

If you forget the command line options run the script with **–help** or visit the [help section](#).

Warning: The **update-cdh**, and **restart-cdh** options are provided as fail-safe to keep unwanted changes from potentially breaking a working cluster, and to keep restarts from stopping long running jobs.

Example

```
$ cluster-config --host CLOUDERA_HOST --port CLOUDERA_PORT --username CLOUDERA_USER --update-cdh yes -
What is the Cloudera manager password?
--INFO cluster selected: SOME_CLUSTER
--INFO Reading CDH config file: /home/some-user/cdh.json
--WARNING Couldn't open json file: /home/rodorad/user-cdh.json
--INFO Writting merged CDH config file: /home/some-user/merged-cdh.json
--INFO Updating config group: REGIONSERVER_BASE
--INFO Updated 1 configuration/s.
Restarting service : "HBASE"
. . . . .

Deploying configuration for all HBASE roles
.

--INFO Updating config group: NODEMANAGER_BASE
--INFO Updated 2 configuration/s.
--INFO Updating config group: GATEWAY_BASE
--INFO Updated 7 configuration/s.
--INFO Updating config group: RESOURCEMANAGER_BASE
--INFO Updated 4 configuration/s.
Restarting service : "YARN"
. . . . .

Deploying configuration for all YARN roles
.
```

After connecting to Cloudera Manager, `cdh.json` (generated in the [previous step](#) and `user-cdh.json` (if available) are read.

The file `user-cdh.json` has any user overrides and/or any additional CDH configurations needing to be set. For this example, the `user-cdh.json` file was omitted.

When **–update-cdh**, and **–restart-cdh** are set to ‘yes’, the CDH needs to update, and restart CDH with the configurations from `cdh.json/user-cdh.json`.

user-cdh.json

The *user-cdh.json* file is where to set any CDH configurations. As long the file is in the working directory or in directory specified by `-path`, it will be read and pushed to CDH by the *cluster-config* script.

A sample *user-cdh.json* file looks like a *cdh.json* file. It can have the same or different keys as a *cdh.json* file.

Finding Keys For a simple demonstration, look for any key with the *cluster-explore* script to add to our *user-cdh.json* file. This example shows **YARN.JOBHISTORY.JOBHISTORY_BASE.YARN_JOBHISTORY_BIND_WILDCARD**.

```
config:
- Name: YARN_JOBHISTORY_BIND_WILDCARD
- Description: If enabled, the JobHistory Server binds to the wildcard address ("0.0.0.0") on all of
- Key: YARN.JOBHISTORY.JOBHISTORY_BASE.YARN_JOBHISTORY_BIND_WILDCARD
- Value: false
```

This is what it looks like in CDH:

Bind JobHistory Server to Wildcard Address JobHistory Server Default Group ☐

Converting Keys To save the key to *user-cdh.json* format, the key needs to be split at the each period with each element being nested inside the previous element.

key: **YARN.JOBHISTORY.JOBHISTORY_BASE.YARN_JOBHISTORY_BIND_WILDCARD** would convert to:

```
{
  "YARN": {
    "JOBHISTORY": {
      "JOBHISTORY_BASE": {
        "YARN_JOBHISTORY_BIND_WILDCARD": "true"
      }
    }
  }
}
```

Lets go ahead and add another key that will conflict with our *cdh.json* file to see how conflict resolution works.

I added **'YARN_NODEMANAGER_RESOURCE_CPU_VCORES'** which is set to 32 cores in my *cdh.json*. Add the key and subtract a couple of cores from the value:

```
{
  "YARN": {
    "JOBHISTORY": {
      "JOBHISTORY_BASE": {
        "YARN_JOBHISTORY_BIND_WILDCARD": "true"
      }
    },
    "NODEMANAGER": {
      "NODEMANAGER_BASE": {
        "YARN_NODEMANAGER_RESOURCE_CPU_VCORES": 30
      }
    }
  }
}
```

```
}
}
```

Once you have your *user-cdh.json* file make sure you save it to your working directory.

Verifying JSON All *user-cdh.json* files need to contain only valid json text. If you get parsing errors or have any doubts about your formatting, you can use a [json lint tool](http://jsonlint.com/)⁸ to format and error check your json text.

Create cdh.json Once you've saved *user-cdh.json*, run *Generating CDH Configurations* to create a *cdh.json* file, or verify you have *_cdh.json_* file in the same working directory where you saved *_user-cdh.json_* file.

Push user-cdh.json Running the *cluster-config* script will automatically push *user-cdh.json* configurations to CDH after merging with *cdh.json*.

You should see log messages notifying you that *user-cdh.json* was loaded.

```
$ cluster-config --host SOME_USER --update-cdh yes --restart-cdh no
What is the Cloudera manager password?
--INFO cluster selected: SOME_CLUSTER
--INFO Reading CDH config file: /home/SOME_USER/cdh.json
--INFO Reading user CDH config file: /home/SOME_USER/user-cdh.json
--INFO conflict resolution: first

Key merge conflict: YARN.NODEMANAGER.NODEMANAGER_BASE.YARN_NODEMANAGER_RESOURCE_CPU_VCORES
[user]User value: 30
[auto]Auto generated value: 32
Optionally you can accept [au] all user values or [ag] all auto generated values.
--INFO Auto resolving conflicts. Defaulting to user value
--INFO Writting merged CDH config file: /home/rodorad/merged-cdh.json
--INFO Updating config group: REGIONSERVER_BASE
--INFO Updated 1 configuration/s.
--INFO Updating config group: NODEMANAGER_BASE
--INFO Updated 2 configuration/s.
--INFO Updating config group: JOBHISTORY_BASE
--INFO Updated 1 configuration/s.
--INFO Updating config group: RESOURCEMANAGER_BASE
--INFO Updated 4 configuration/s.
--INFO Updating config group: GATEWAY_BASE
--INFO Updated 7 configuration/s.
```

In above example we see notifications when we load *user-cdh.json* and how conflicts were resolved.

The default behavior when encountering conflicting keys is to resolve them to the user value. You can change the default behavior on the command line with the **–conflict-merge** option.

We also see that a *merged-cdh.json* file was created. This file gets created when the *user-cdh.json* and *cdh.json* files are merged. It is a record of all the configurations that got pushed to CDH after resolving key conflicts.

After pushing our changes our CDH value is updated.

Bind JobHistory Server to Wildcard Address

JobHistory Server Default Group ☒ 

⁸<http://jsonlint.com/>

merged-cdh.json sample merged-cdh.json file

```
{
  "HBASE": {
    "REGIONSERVER": {
      "REGIONSERVER_BASE": {
        "HBASE_REGIONSERVER_JAVA_HEAPSIZE": 26430567975
      }
    }
  },
  "YARN": {
    "GATEWAY": {
      "GATEWAY_BASE": {
        "MAPREDUCE_MAP_JAVA_OPTS_MAX_HEAP": 4129776246,
        "MAPREDUCE_MAP_MEMORY_MB": 5368709120,
        "MAPREDUCE_REDUCE_JAVA_OPTS_MAX_HEAP": 8259552492,
        "MAPREDUCE_REDUCE_MEMORY_MB": 10737418240,
        "YARN_APP_MAPREDUCE_AM_MAX_HEAP": 4129776246,
        "YARN_APP_MAPREDUCE_AM_RESOURCE_CPU_VCORES": 1,
        "YARN_APP_MAPREDUCE_AM_RESOURCE_MB": 5368709120
      }
    },
    "JOBHISTORY": {
      "JOBHISTORY_BASE": {
        "YARN_JOBHISTORY_BIND_WILDCARD": "True"
      }
    },
    "NODEMANAGER": {
      "NODEMANAGER_BASE": {
        "YARN_NODEMANAGER_RESOURCE_CPU_VCORES": 30,
        "YARN_NODEMANAGER_RESOURCE_MEMORY_MB": 180388626432
      }
    },
    "RESOURCEMANAGER": {
      "RESOURCEMANAGER_BASE": {
        "YARN_SCHEDULER_INCREMENT_ALLOCATION_MB": 536870912,
        "YARN_SCHEDULER_MAXIMUM_ALLOCATION_MB": 180388626432,
        "YARN_SCHEDULER_MAXIMUM_ALLOCATION_VCORES": 32,
        "YARN_SCHEDULER_MINIMUM_ALLOCATION_MB": 5368709120
      }
    }
  }
}
```

Exploring CDH Configurations

Script: **cluster-explore**

Description: Browse CDH configurations by service, role, and configuration group or dumped all at once to the screen. Helps find the configuration keys if you want to write your own *user-cdh.json* file or if you want to write your own formula file.

Unique Options:

- **-dump**: if 'yes' dumps all the configurations for every service to stdout.

Common Options

If you forget the command line options run the script with **-help** or visit the [help section](#).

Finding configurations is quite difficult without an appropriate tool. The cluster explore script solves this very problem.

It has two run modes:

1. Print all the configurations for every service to the screen.
2. Interactively print all the configurations for a specific service, role, and configuration group.

To invoke option 1 you can answer 'yes' to the **-dump** command line option or answer yes when prompted on the command line.

```
$ cluster-explore --host SOME_HOST --port SOME_PORT --username SOME_USERNAME
What is the Cloudera manager password?
--INFO cluster selected: SOME_CLUSTER
dump all configs[yes or no]: yes

...
- config name: ENABLE_CONFIG_ALERTS
- config description: When set, Cloudera Manager will send alerts when this entity's configuration ch
- config key: SPARK_ON_YARN.SPARK_YARN_HISTORY_SERVER.SERVER_BASE.ENABLE_CONFIG_ALERTS
- config value: false

- config name: PROCESS_AUTO_RESTART
- config description: When set, this role's process is automatically (and transparently) restarted in
- config key: SPARK_ON_YARN.SPARK_YARN_HISTORY_SERVER.SERVER_BASE.PROCESS_AUTO_RESTART
- config value: false
...
```

To invoke option 2, run the script interactively. Answer “no” or hit enter when prompted to dump the configurations.

```
$ cluster-explore --host SOME_HOST --port SOME_PORT --username SOME_USERNAME
What is the Cloudera manager password?
--INFO cluster selected: SOME_CLUSTER
dump all configs[yes or no]: no

Available service types on cluster: 'cluster'
Pick a service
Id 1 service: HDFS
Id 2 service: HBASE
Id 3 service: ZOOKEEPER
Id 4 service: YARN
Id 5 service: SPARK_ON_YARN
Enter service Id : 4
Selected YARN

Available role types on service: 'YARN'
Pick a role
Id 1 role: NODEMANAGER
Id 2 role: JOBHISTORY
Id 3 role: RESOURCEMANAGER
Id 4 role: GATEWAY
Enter role Id : 4
Selected GATEWAY

Available config group types on role: 'GATEWAY'
```

```

Pick a config group
Id 1 config group: GATEWAY_BASE
Enter config group Id : 1
Selected GATEWAY_BASE

...

config:
- name: MAPRED_OUTPUT_COMPRESSION_CODEC
- description: For MapReduce job outputs that are compressed, specify the compression codec to use. W
- key: YARN.GATEWAY.GATEWAY_BASE.MAPRED_OUTPUT_COMPRESSION_CODEC
- value: org.apache.hadoop.io.compress.DefaultCodec

config:
- name: MAPRED_REDUCE_PARALLEL_COPIES
- description: The default number of parallel transfers run by reduce during the copy (shuffle) phase
- key: YARN.GATEWAY.GATEWAY_BASE.MAPRED_REDUCE_PARALLEL_COPIES
- value: 10

config:
- name: MAPREDUCE_JOB_UBERTASK_MAXMAPS
- description: Threshold for number of maps, beyond which a job is considered too big for ubertask op
- key: YARN.GATEWAY.GATEWAY_BASE.MAPREDUCE_JOB_UBERTASK_MAXMAPS
- value: 9

...

```

For every configuration key you will see

- configuration name: Normalized configuration name to account for some weirdness with CDH configuration names like slashes and dashes.
- configuration description: This will be the description from CDH UI.
- full configuration key path: The format for a key is `SERVICE_TYPE.ROLE_TYPE.CONFIG_GROUP_NAME.CONFIGURATION_KEY`. While composed mostly of types the `CONFIG_GROUP_NAME` is normalized to remove service names which can be anything.
- value: The value of the configuration.

14.1.2 Giraph Performance Tuning and Benchmarking

Apache Giraph is a Hadoop-based graph-processing framework that is modeled after Google's Pregel. Giraph jobs are memory-hungry applications because it attempts to load the entire graph in-memory. Giraph jobs have no reducers. To improve memory-related issues:

- Allocate all available memory to the job, in other words, set the total JVM heap size available to mappers in `$HADOOP_HOME/conf/mapred-site` (`mapreduce.tasktracker.map.tasks.maximum` * maximum heap size in `mapred.map.child.java.opts`) so it does not exceed available memory for the node, and leaves room for other daemons running on the nodes, for example, datanode, regionserver, and tasktracker.
- Allocate at least 4GB to each map tasks, but this value depends heavily on the size of the graph.
- The maximum number of workers available is the number of map slots available in the cluster - 1 (in other words, the number of nodes times the `mapreduce.tasktracker.map.tasks.maximum` - 1). One master and the rest are slaves.
- When writing Giraph jobs, keep the compute function small and minimize vertex memory footprint. There is a patch for spilling data to disk but it does not appear to be working very well at the moment.

To solve performance problems, try upgrading to the latest version of Giraph, which might include patches that address performance concerns. For example, Facebook submitted a large number of patches that allow Giraph to run at scale [http link](#)⁹. Some memory optimizations to try are:

- Enable ByteArrayPartition store that Facebook implemented to optimize memory usage in Giraph using the option **-Dgiraph.partitionClass=org.apache.giraph.partition.ByteArrayPartition**. This option significantly decreases the amount of memory needed to run Giraph jobs. Giraph's default partition store is the SimplePartitionStore (which stores the graph as Java objects). The ByteArrayPartition store keeps only a representative vertex that serializes/deserializes on the fly thereby saving memory. Be careful to use this option, because some algorithms may fail with this option on.
- If messages are stored as complex objects, try enabling the option **-Dgiraph.useMessageSizeEncoding=true**.
Note: This option will cause jobs to crash if the messages are simple objects.

Other Giraph options are available at <http://giraph.apache.org/options.html>. These options can be set using **-D<option=value>** in the command for running the Hadoop job. For large clusters, “The Hadoop Real World Solutions Cookbook, Owens et al. 2013” recommends controlling the number of messaging threads used by each worker to communicate with other workers, and increasing the maximum number of messages flushed in bulk.

14.1.3 HBase Performance Tuning

Two comprehensive sources for performance tuning in HBase are:

- [HBase reference guide](#)¹⁰
- [The HBase Administration Cookbook](#)¹¹

Some of the recommendations for production clusters are detailed below. Remember to sync these changes to all servers in the cluster.

1. Increase ulimit settings. The default Linux setting for maximum number of file descriptors for a user is 1024 which is too low. The number of file descriptors (nofile) and open processes for the user running the Hadoop/HBase daemons should be set to a minimum of 32768, otherwise the system will experience problems under heavy load. Instructions for changing */etc/security/limits.conf* are available here: <http://hbase.apache.org/book.html#ulimit>
2. Garbage collection. Pauses due to garbage collection can impact performance. For latency-sensitive applications, Cloudera recommends **-XX:+UseParNewGC** and **-XX:+UseConcMarkSweepGC** in *hbase-env.sh*. The MemStore-Local Allocation buffers (MSLAB) which reduce heap fragmentation under heavy write-load are enabled by default in HBase. The MSLAB setting is **hbase.hregion.memstore.mslab.enabled** in *hbase-site.xml*. <http://blog.cloudera.com/blog/2011/02/avoiding-full-gcs-in-hbase-with-memstore-local-allocation-buffers-part-1>
3. Enable column family compression. Compression boosts performance by reducing the size of store files thereby reducing I/O <http://hbase.apache.org/book.html#compression>. The Snappy library from Google is faster than LZO and easier to install. Instructions for installing Snappy are available at: <http://hbase.apache.org/book.html#d0e24641> Once installed, turn on column family compression when creating the table, for example:

```
create 'mytable', {NAME => 'mycolumnfamily1', COMPRESSION => 'SNAPPY'}
```

4. Avoid update blocking for write-heavy workloads. If the cluster has enough memory, increase **hbase.hregion.memstore.block.multiplier** and **hbase.hstore.blockingStoreFiles** in *hbase-site.xml* to avoid

⁹<http://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920>

¹⁰http://hbase.apache.org/book.html#important_configurations

¹¹<http://library.intel.com/Catalog/CatalogItemDetails.aspx?id=225983>

blocking in write-heavy workloads. This setting needs to be tuned carefully because it increases the number of files to compact. See [The HBase Administration Cookbook](#)¹² (Chapter 9)

5. Increase region server handler count. If the workload has many concurrent clients, increasing the number of handlers can improve performance. However, too many handlers will lead to out of memory errors. See [The HBase Administration Cookbook](#)¹³ (Chapter 9)

Other recommendations include turning off automatic compactions and doing it manually:
<http://hbase.apache.org/book.html#managed.compactions>.

¹²<http://library.intel.com/Catalog/CatalogItemDetails.aspx?id=225983>

¹³<http://library.intel.com/Catalog/CatalogItemDetails.aspx?id=225983>

Part IV

Administration

SYSTEM REQUIREMENTS

15.1 Physical Machine Configuration

Table of Contents

- *Pre-Configuration*
- *Base Configuration*
 - *1. Client-side DNS (Domain Name Service)*
 - *2. Firewall*
 - *3. SELINUX*
 - *4. Default Repositories*
 - *5. Proxy Settings*
 - *6. Synchronize System Packages*
 - *7. Primary CDH User*
 - *8. ssh Connections*
 - *9. System Hostname*
 - *10. Ulimits*
 - *11. NTP*
 - *12. Reboot*

The following tutorial goes through configuring a new physical machine from scratch. The first section, called “Pre-Configuration”, is a little vague because the information can vary between machines. The second section, called “Base Configuration”, will usually be identical for all machines.

15.1.1 Pre-Configuration

1. Configure basic network connectivity.
2. Turn off IPV6 (many different ways to do this).
3. If applicable, mount any large volume to /mnt* (hint: vi /etc/fstab).

15.1.2 Base Configuration

1. Client-side DNS

All systems in the cluster must be reachable by DNS or by data in ‘etc/hosts’.

2. Firewall

The firewall must be disabled:

```
service iptables stop
chkconfig iptables off
```

3. SELINUX

SELINUX must be disabled:

```
vi /etc/selinux/config
```

Change:

```
SELINUX=enforcing
```

To:

```
SELINUX=disabled
```

Or:

```
SELINUX=permissive
```

Note: Systems will need full reboot before changes take effect.

4. Default Repositories

System default yum repos must be functional.

5. Proxy Settings

If working behind a proxy, system proxy settings must be configured.

6. Synchronize System Packages

System packages must be synchronized with default repositories:

```
yum clean all
yum distro-sync
```

7. Primary CDH User

Cloudera supports use of root or sudo user as administration user. If using sudo, user must have full nopassword sudo privileges.

8. *ssh* Connections

Using the primary CDH user, every system in the cluster must be able to communicate via *ssh* to all other systems in the cluster.

9. System Hostname

Set the hostname for each system in the cluster.

Note: Limiting host names to lower-case alphanumeric characters is recommended.

10. Ulimits

The following definitions must exist in `/etc/security/limits.conf`

```
vi /etc/security/limits.conf

*                soft    nofile           32768
*                hard    nofile           32768
hadoop           -       nofile           32768
hadoop           -       nproc            unlimited
hdfs             -       nofile           32768
hbase            -       nofile           32768
spark            soft    nofile           65535
spark            hard    nofile           65535
spark            -       nproc            32768
```

11. NTP

NTP must be installed and properly configured on all cluster systems. Also NTP services should start on system boot:

```
service ntpd start
chkconfig ntpd on
```

All systems in cluster must be in time-sync with one-another.

12. Reboot

Once all configuration has been done, rebooting all cluster systems is recommended to properly activate all of the changes.

15.2 Cloudera Hadoop 5 Configuration

Table of Contents

- *Install Cloudera Manager*
- *Proxy and Parcel Info in Cloudera Manager*
- *Submit License File*
- *Specifying Hosts*
- *CDH Parcel Repository*
- *Java Encryption*
- *SSH Login Credentials*
- *Cluster Installation*
- *Host Configuration*
- *CDH Services to Install*
- *Customize Role Assignments*
- *Database Setup*
- *Review Changes*
- *Finishing Up In Cloudera Manager*
- *Final Settings and Tests*
- *Tweaks*

This guide discusses the process of configuring Cloudera Hadoop 5 on a physical or virtual cluster.

15.2.1 Install Cloudera Manager

The Cloudera Manager must be downloaded and installed. For instructions, see the Cloudera website (http://www.cloudera.com/content/support/en/downloads/cloudera_manager/cm-5-1-0.html).

15.2.2 Proxy and Parcel Info in Cloudera Manager

1. On a web browser, go to the Cloudera Manager.
2. Click the *Cloudera Manager* hyperlink graphic on the top left portion of the window
3. Click the *Administration* drop-down along the top of the window, then select *Settings*
4. Select the *Network* button along the menu pane to the left
5. In the *Proxy Server* field, enter the proxy qualified name, for example, `proxy.my.company.com`
6. In the *Proxy Port* field, enter the proxy port number
7. Select the *Parcels* button along the menu pane to the left
 - (a) Overwrite the field that says `http://archive.cloudera.com/cdh5/parcels/latest/` with `http://archive.cloudera.com/cdh5/parcels/5.3.1/`
8. Hit the *Save Changes* button to the top right of the active menu
9. Hit the admin drop-down menu at the top right corner of the window and logout
10. Log back in using the same admin username password combo

15.2.3 Submit License File

1. Acquire the Cloudera license file.
2. Under the *Cloudera Enterprise* column, click on the empty text field to the left of the Upload button

3. Select the license file
4. Hit the *Upload* button
5. Hit *Continue* on the bottom right of the window

15.2.4 Specifying Hosts

This step connects the master node to the rest of the cluster. The syntax used to search for hostnames is identical to what can be found in the `/etc/hosts` file or by DNS lookup.

Hit *Continue* through the “Thank you for choosing Cloudera Manager and CDH” window. In the text field presented, enter the hostnames of each node in the following syntax:


```
master.clustername.cluster
node[01-03].clustername.cluster
```

Where `clustername` is the name of the cluster, and `[01-03]` is the range of slave nodes in the cluster (`[01-07]` for an 8 node cluster, `[01-15]` for a 16 node cluster).

Hit *Search*, and make sure that the computer detects as many hosts as there are nodes in the cluster. See [Fig. 11.1](#) for examples. If all correct hosts are selected, hit *Continue*. Otherwise, click *New Search*.

Hosts should be specified using the same hostname (FQDN) that they will identify themselves with.

Cloudera recommends including Cloudera Manager Server's host. This will also enable health monitoring for that host.

Hint: Search for hostnames and/or IP addresses using [patterns](#) .

4 hosts scanned, 4 running SSH.
Q New Search

<input checked="" type="checkbox"/>	Expanded Query	Hostname (FQDN)	IP Address	Currently Managed	Result
<input checked="" type="checkbox"/>	master.khaos.cluster	master.khaos.cluster	10.0.130.41	No	✓ Host ready: 0 ms re
<input checked="" type="checkbox"/>	node01.khaos.cluster	node01.khaos.cluster	10.0.130.166	No	✓ Host ready: 3 ms re
<input checked="" type="checkbox"/>	node02.khaos.cluster	node02.khaos.cluster	10.0.130.135	No	✓ Host ready: 3 ms re
<input checked="" type="checkbox"/>	node03.khaos.cluster	node03.khaos.cluster	10.0.131.253	No	✓ Host ready: 4 ms re

Fig. 15.1: Specify hosts for your CDH cluster installation.

15.2.5 CDH Parcel Repository

The repository/proxy information should populate the parcel list in a minute. If not, click on *More Options* field to reconfigure. Make sure `CDH-5.3.1-1.cdh5.3.1.p`.3 is selected under *Remote Parcel Repository* (see [Fig. 11.2](#)) and then hit *Continue*.

Cloudera recommends the use of parcels for installation over packages, because parcels enable Cloudera Manager to easily manage the software on your cluster, automating the deployment and upgrade of service binaries. Electing not to use parcels will require you to manually upgrade packages on all hosts in your cluster when software updates are available, and will prevent you from using Cloudera Manager's rolling upgrade capabilities.

- Choose Method**
- ☐ Use Packages ?
 - ☒ Use Parcels (Recommended) ?

Select the version of CDH

- ☒ CDH-5.0.2-1.cdh5.0.2.p0.13
- ☐ CDH-4.7.0-1.cdh4.7.0.p0.40

- Additional Parcels**
- ☐ ACCUMULO-1.4.4-1.cdh4.5.0.p0.65
 - ☒ None

- ☐ SQOOP_NETEZZA_CONNECTOR-1.2c5
- ☒ None

- ☐ SQOOP_TERADATA_CONNECTOR-1.2c5
- ☒ None

More Options

+ Add

Example ?

`http://archive.cloudera.com/cdh5/parcels/latest`

Select the specific release of the Cloudera Manager Agent you want to install on your hosts.

- ☒ Matched release for this Cloudera Manager Server
- ☐ Custom Repository

Fig. 15.2: Select Repository

15.2.6 Java Encryption

Java encryption is not currently supported.

15.2.7 SSH Login Credentials

Fill out appropriate login information for CDH administrator user.

15.2.8 Cluster Installation

The next couple of windows are just progress bars. If any of them fail and turn red, sometimes just hitting *Retry* will fix the problem nodes. See [Fig 11.3](#).

Hit *Continue* button when it lights up after the progress bar fills. You will be greeted by more progress bars. Wait and hit *Continue* when they finish too.

Installation in progress.

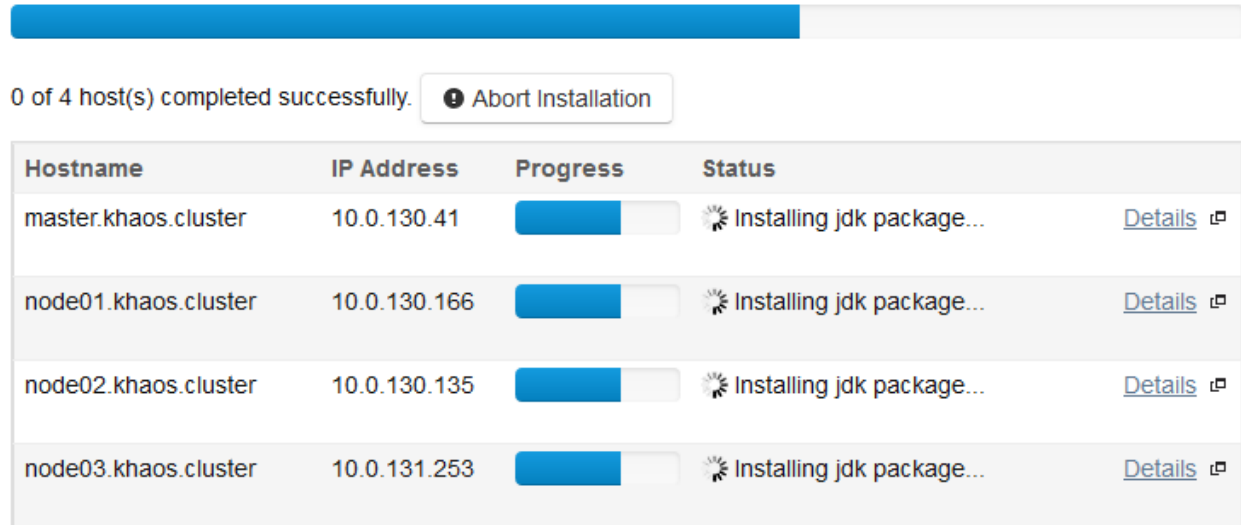


Fig. 15.3: Cluster Installation

15.2.9 Host Configuration

When the cluster installation finishes, look for any critical errors. Take note of anything that doesn't have a green check mark next to it and resolve the issue. See [Fig. 11.4](#).

Click *Finish*

15.2.10 CDH Services to Install

Choose the CDH 5 services to install on your cluster. The following windows will show the process of installing services and roles on each node in the cluster. This is the Trusted Analytics default setup.

In the "Choose a combination of services to install" dialogue, select the "Custom Services" button. In the drop-down menu, mark the following boxes:

- ✓ Inspector ran on all 4 hosts.
- ✓ The following failures were observed in checking hostnames...
- ✓ No errors were found while looking for conflicting init scripts.
- ✓ No errors were found while checking /etc/hosts.
- ✓ All hosts resolved localhost to 127.0.0.1.
- ✓ All hosts checked resolved each other's hostnames correctly and in a timely manner.
- ✓ Host clocks are approximately in sync (within ten minutes).
- ✓ Host time zones are consistent across the cluster.
- ✓ No users or groups are missing.
- ✓ No conflicts detected between packages and parcels.
- ✓ No kernel versions that are known to be bad are running.
- ✓ All hosts have /proc/sys/vm/swappiness set to 0.
- ⚠ Transparent Huge Pages is enabled and can cause significant performance problems. Kernel with release 'Red Hat Enterprise Linux Server release 6.4 (Santiago)' and version '2.6.32-358.11.1.el6.x86_64' has enabled set to '[always] never' and defrag set to '[always] never'. Run "echo never > /sys/kernel/mm/redhat_transparent_hugepage/defrag" to disable this, then add the same command to an init script such as /etc/rc.local so it will be set upon system reboot. Alternatively, upgrade to RHEL 6.5 or later, which does not have this bug. The following hosts are affected: ➤
- ✓ 0 hosts are running CDH 4 and 4 hosts are running CDH5.
- ✓ All checked hosts in each cluster are running the same version of components.
- ✓ All managed hosts have consistent versions of Java.
- ✓ All checked Cloudera Management Daemons versions are consistent with the server.
- ✓ All checked Cloudera Management Agents versions are consistent with the server.

Fig. 15.4: Host Configuration

- HBase
- HDFS
- Spark
- YARN (MR2 Included)
- ZooKeeper

See [Fig. 11.5](#). Click *Continue*.

15.2.11 Customize Role Assignments

This page allows designation of which roles the different nodes will take up. In a default loadout, almost all of these fields will be left to their default, but there are four that need to be changed.

1. Under the HBase section, click on the *HBase Thrift Server* dialogue and select the “master” node of the cluster
2. Under the HDFS section, click on the *Secondary Name Node* dialogue and select “node01” of the cluster
3. Under the YARN section, click on the *Job History Server* dialogue and select “node01” of the cluster
4. Under the *ZooKeeper* section, click on the *Server* dialogue and select “node01”, “node02” and “node03” of the cluster

Leave all other fields in their default values and click *Continue*.

See [Fig. 11.6](#) for changes to make near the top:

See [Fig. 11.7](#) for changes to make near the bottom:

15.2.12 Database Setup

The “Database Host Name” field should auto-populate with the hostname of the system on which Cloudera Manager is installed. If not, fill that in.

Click *Test Connection*. See [Fig 11.8](#). If successful, click *Continue*.

15.2.13 Review Changes

In the “Review Changes” window, all fields should remain their default values.

Click *Continue*.














15.2.14 Finishing Up In Cloudera Manager

The next page requires no interaction. Just more loading bars.

1. Wait for all services to start up, then hit *Continue*.
2. In the *Congratulations!* window, click *Finish*.
3. Some of the health indicators may be orange or red in the first few moments of the cluster’s life. Wait a minute for them to all turn green.
4. In the Cloudera Manager page, change the name of the cluster by hitting the drop down arrow to the right of the *Cluster 1* heading then clicking *Rename Cluster*. See [Fig. 11.9](#).

Custom Services

Choose your own services. Services required by chosen services will automatically be included. Note that Flume can be added after your initial cluster has been set up.

Service Type	Description
<input checked="" type="checkbox"/>  HBase	Apache HBase provides random, real-time, read/write access to large data sets (requires HDFS and ZooKeeper).
<input checked="" type="checkbox"/>  HDFS	Apache Hadoop Distributed File System (HDFS) is the primary storage system used by Hadoop applications. HDFS creates multiple replicas of data blocks and distributes them on compute hosts throughout a cluster to enable reliable, extremely rapid computations.
<input type="checkbox"/>  Hive	Hive is a data warehouse system that offers a SQL-like language called HiveQL.
<input type="checkbox"/>  Hue	Hue is a graphical user interface to work with Cloudera's Distribution Including Apache Hadoop (requires HDFS, MapReduce, and Hive).
<input type="checkbox"/>  Impala	Impala provides a real-time SQL query interface for data stored in HDFS and HBase. Impala requires Hive service and shares Hive Metastore with Hue.
<input type="checkbox"/>  Key-Value Store Indexer	Key-Value Store Indexer listens for changes in data inside tables contained in HBase and indexes them using Solr.
<input type="checkbox"/>  MapReduce	Apache Hadoop MapReduce supports distributed computing on large data sets across your cluster (requires HDFS). YARN (includes MapReduce 2) is recommended instead. MapReduce is included for backwards-compatibility.
<input checked="" type="checkbox"/>  Oozie	Oozie is a workflow coordination service to manage data processing jobs on your cluster.
<input type="checkbox"/>  Solr	Solr is a distributed service for indexing and searching data stored in HDFS.
<input checked="" type="checkbox"/>  Spark	Apache Spark is an open source cluster computing system
<input checked="" type="checkbox"/>  Sqoop 2	Sqoop is a tool designed for efficiently transferring bulk data between Apache Hadoop and structured datastores such as relational databases. The version supported by Cloudera Manager is Sqoop 2 .
<input checked="" type="checkbox"/>  YARN (MR2 Included)	Apache Hadoop MapReduce 2.0 (MRv2), or YARN, is a data computation framework that supports MapReduce applications (requires HDFS).
<input checked="" type="checkbox"/>  ZooKeeper	Apache ZooKeeper is a centralized service for maintaining and synchronizing configuration data.

Note: Please ensure that you have the appropriate license for **HBase** or contact Cloudera for assistance.

Fig. 15.5: Custom CDH Services

The Hbase configuration interface includes several components:

- Master**: 1 New, master
- HBase REST Server**: Select hosts
- HBase Thrift Server**: master (highlighted with a red box)
- RegionServer**: Same As DataNode
- HDFS**:
 - NameNode**: 1 New, master
 - SNN**: node01 (highlighted with a red box)
 - HttpFS**: Select hosts
 - NFS Gateway**: Select hosts
- DataNode**: 3 New, node[01-03]

Fig. 15.6: Hbase

The Yarn configuration interface includes several components:

- ResourceManager**: master
- JobHistory Server**: node01 (highlighted with a red box)
- NodeManager**: Same As DataNode
- ZooKeeper**:
 - Server**: 3 New, node[01-03] (highlighted with a red box)

Fig. 15.7: Yarn

Database Setup

Configure and test database connections. If using custom databases, create the databases first according to the **Installing and Configuring an External Database** section of the [Installation Guide](#) .

☐ Use Custom Databases
☒ Use Embedded Database

When using the embedded database, passwords are automatically generated. Please copy them down.

Reports Manager ✓ Successful

Currently assigned to run on **master.khaos.cluster**.

Database Host Name:	Database Type:	Database Name :	Username:	Password:
master.khaos.cluster:7432	PostgreSQL	rman	rman	LJW3HXj4EC

[Test Connection](#)

Fig. 15.8: Database Setup

5. In the Cloudera Manager, hit the admin drop-down at the top right corner of the screen and select *Change Password*. Change the password as desired.
6. Select the Spark service from the homescreen.
 - (a) Select *Configuration* along the top Spark menu.
 - (b) Select *Worker Default Group* along the left side menu pane.
 - (c) Select the *Work Directory* field and change the value to a directory with the capacity to store lots of temporary data (the /mnt directory for virtual clusters).

15.2.15 Final Settings and Tests

Test functionality of HDFS.

15.2.16 Tweaks

The graph machine learning algorithms in Trusted Analytics use the Giraph graph-processing framework. Giraph is designed to run the whole graph computation in memory, and requires large amounts of memory to process big graphs. There should be at least 4GB of memory per map task to cater for graphs with supernodes. Giraph jobs are scheduled using YARN. If a Giraph job requests twice the amount of memory configured in YARN, then the YARN resource manager will not schedule it causing the job to hang.

To run Giraph jobs, ensure that the memory settings in CDH match those in application.conf using one of the following approaches:

1. Modify the following YARN configuration in CDH to match the setting under trustedanalytics.atk.giraph in application.conf. Under the YARN section in CDH, click on *Configuration* and select *View and Edit*.

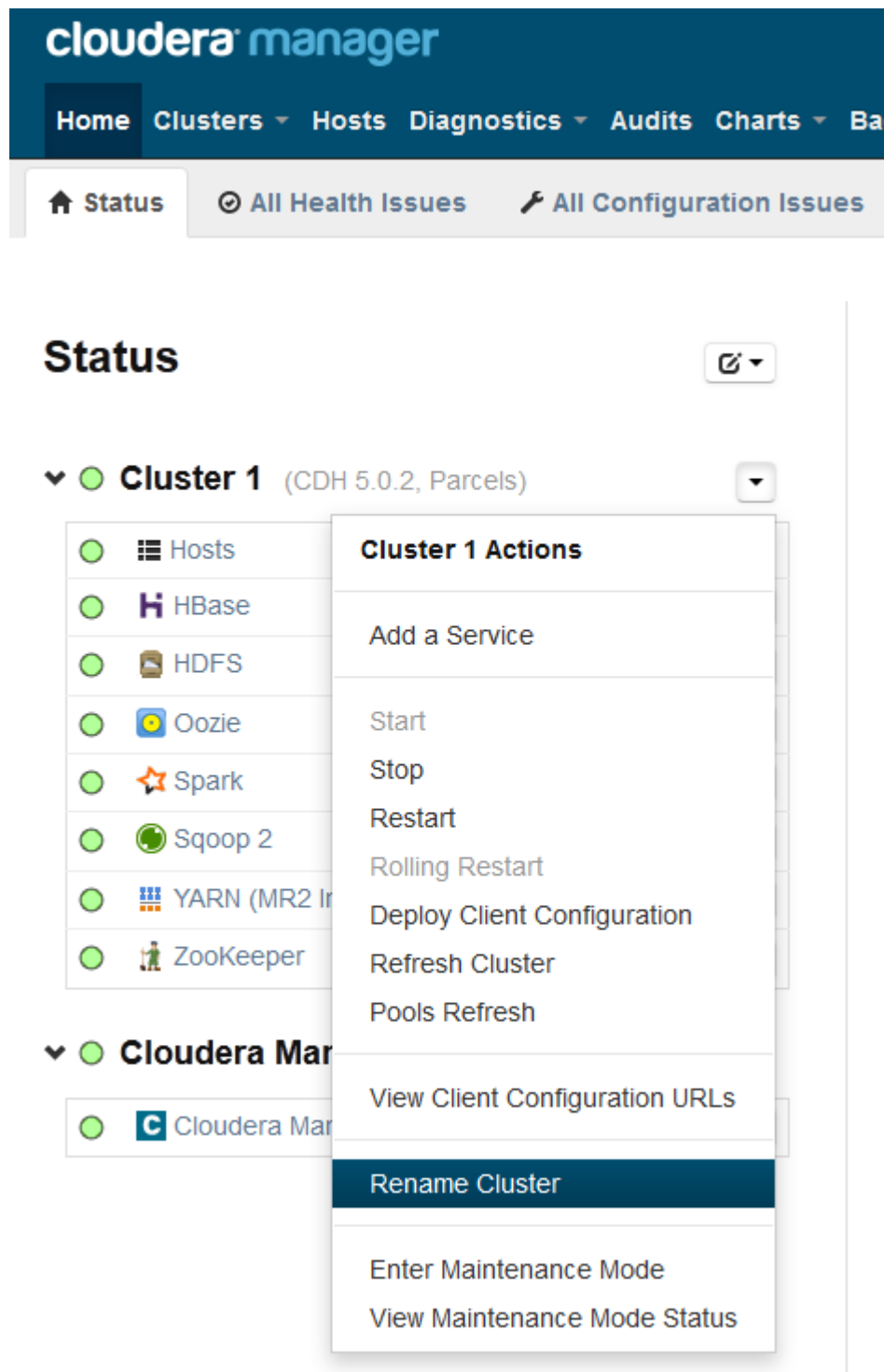


Fig. 15.9: Finishing Up In Cloudera Manager

- (a) Search for `mapreduce.map.memory.mb` in the search box on the upper left corner. Modify `mapreduce.map.memory.mb` to match `mapreduce.map.memory.mb` in `application.conf` (currently 4096 MB)
 - (b) Search for `mapreduce.map.java.opts.max` in the search box. Modify this setting to match `mapreduce.map.java.opts` in `application.conf` (currently 3072MB). The rule of thumb is that `mapreduce.map.java.opts.max` should be at most 85% of `mapreduce.map.memory.mb`
 - (c) Search for `yarn.nodemanager.resource.memory-mb` in the search box. Modify this setting to a multiple of `mapreduce.map.memory.mb`. For example, to run at most 4 mappers on each node, and `mapreduce.map.memory.mb` is set to 4096MB, then set `yarn.nodemanager.resource.memory-mb` to 16384MB.
 - (d) Save these changes.
 - (e) Click on *Actions*, on the top-right corner and then *Deploy Client Configuration* to update the configurations across the cluster.
 - (f) Restart YARN.
2. Limit the Giraph memory allocation in `application.conf` to match the configured CDH settings in YARN. The relevant settings in the Trusted Analytics `application.conf` file are in `trustedanalytics.atk.giraph`:
 - (a) `mapreduce.map.memory.mb`. This setting should match `mapreduce.map.memory.mb` in YARN.
 - (b) `mapreduce.map.java.opts`. This setting should match `mapreduce.map.java.opts.max` in YARN.
 - (c) `giraph.maxWorkers`. The maximum value for this setting should be the maximum number of map tasks that can run on the cluster - 1. One mapper is reserved for the Giraph master, while the rest of the mappers are Giraph workers. Since Giraph is memory-intensive, a good estimate for `giraph.maxWorkers` is $((\text{Number of Yarn node managers} * \text{yarn.nodemanager.resource.memory-mb} / \text{yarn.nodemanager.resource.memory-mb}) - 1)$.

The hardware requirements for the Trusted Analytics package server are insignificant with comparison to the requirements for a Cloudera server. See [Cloudera Requirements](#)¹.

Requirements specifically for the Trusted Analytics Package can be found in [Package Installation](#).

¹<http://blog.cloudera.com/blog/2013/08/how-to-select-the-right-hardware-for-your-new-hadoop-cluster/>

PACKAGE INSTALLATION

- *Introduction*
- *Requirements*
 - *Operating System Requirements*
 - *Cluster Requirements*
- *Trusted Analytics Package Installation*
 - *Adding Extra Repositories*
 - * *EPEL Repository*
 - * *Trusted Analytics Dependency Repository*
 - * *Trusted Analytics Private Repository*
 - *Troubleshooting Private Repository*
 - *Installing Trusted Analytics Package*
 - * *Installing On The Master Node*
 - * *Installing On A Worker Node*
- *REST Server Configuration*
 - *Configuration Script*
 - *Manual Configuration*
 - * */etc/trustedanalytics/rest-server/application.conf*
 - *Configure File System Root*
 - *Configure Zookeeper Hosts*
 - *Configure Spark Master Host*
 - *Configure Spark Executor Memory*
 - *Set the Bind IP Address (Optional)*
 - *Updating the Spark Class Path*
 - *Database Configuration*
 - * *H2*
 - * *PostgreSQL*
 - *Starting The Trusted Analytics REST Server*
 - *Troubleshooting Trusted Analytics REST Server*
 - *Upgrading*

16.1 Introduction

This guide covers Trusted Analytics installation and configuration.

Cloudera installation documentation can be found at: http://www.cloudera.com/content/cloudera-content/cloudera-docs/CM5/latest/Cloudera-Manager-Installation-Guide/cm5ig_install_cm_cdh.html

16.2 Requirements

16.2.1 Operating System Requirements

These instructions are oriented towards [Red Hat Enterprise Linux](http://redhat.com/)¹ or [CentOS](http://centos.org/)² version 6.6. Trusted Analytics uses ‘yum’ for installation, ‘sudo’ for proper authority.

16.2.2 Cluster Requirements

Cloudera cluster 5.3.x with following services:

1. HDFS
2. SPARK
3. Hbase
4. Yarn(MR2)
5. Zookeeper

Trusted Analytics Python client supports Python 2.7.

16.3 Trusted Analytics Package Installation

16.3.1 Adding Extra Repositories

The EPEL and Trusted Analytics repositories must be installed on the REST server node and all spark nodes (master and worker). The Trusted Analytics Dependency repository and the yum-s3 package must be installed before the Trusted Analytics private repository.

EPEL Repository

Verify the installation of the “epel” repository:

..code:

```
$ sudo yum repolist
```

Sample output:

repo id	repo name
cloudera-cdh5	Cloudera Hadoop, Version 5 ...
cloudera-manager	Cloudera Manager, Version 5.3.1 ...
epel	Extra Packages for Enterprise Linux 6 ...
rhui-REGION-client-config-ser...	Red Hat Update Infrastructure 2.0 Clie...
rhui-REGION-rhel-server-relea...	Red Hat Enterprise Linux Server 6 (RPM...
rhui-REGION-rhel-server-relea...	Red Hat Enterprise Linux Server 6 Opti...

If the “epel” repository is not listed, do this to install it:

¹<http://redhat.com/>

²<http://centos.org/>

```
$ wget http://download.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.
noarch.rpm
$ sudo rpm -ivh epel-release-6-8.noarch.rpm
```

Trusted Analytics Dependency Repository

Some open source libraries are included to aid with the installation of the Trusted Analytics. Some of these libraries are newer versions than what is available in RHEL, EPEL or CentOS repositories.

To add the dependency repository, do this:

```
$ wget https://trustedanalytics-dependencies.s3-us-west-2.amazonaws.com/
ta-deps.repo
$ sudo cp ta-deps.repo /etc/yum.repos.d/
```

Alternatively, do this to build the dependency repository information file directly:

```
$ echo "[trustedanalytics-deps]
> name=trustedanalytics-deps
> baseurl=https://trustedanalytics-dependencies.s3-us-west-2.amazonaws.com/yum
> gpgcheck=0
> priority=1 enabled=1" | sudo tee -a /etc/yum.repos.d/ta-deps.repo
```

Test the installation of the dependencies repository:

```
$ sudo yum info yum-s3
```

Results should be similar to this:

```
Available Packages
Name       : yum-s3
Arch       : noarch
Version    : 0.2.4
Release    : 1
Size       : 9.0 k
Repo       : trustedanalytics-deps
Summary    : Amazon S3 plugin for yum.
URL        : git@github.com:NumberFour/yum-s3-plugin.git
License    : Apache License 2.0
```

Installing the `yum-s3` package allows access to the Amazon S3 repository. To install the `yum-s3` package, do this:

```
$ sudo yum -y install yum-s3
```

Trusted Analytics Private Repository

Create `/etc/yum.repos.d/ta.repo`:

```
$ echo "[trustedanalytics]
> name=trustedanalytics
> baseurl=https://trustedanalytics-repo.s3-us-west-2.amazonaws.com/
  release/latest/yum/dists/rhel/6
> gpgcheck=0
> priority=1
> s3_enabled=1
> key_id=ACCESS_TOKEN
> secret_key=SECRET_TOKEN" | sudo tee -a /etc/yum.repos.d/ta.repo
```

Note: baseurl line above is broken for readability. It should be entered as a single line.

Note: Replace “ACCESS_TOKEN” and “SECRET_TOKEN” with appropriate tokens.

To verify the installation of the Trusted Analytics repository, do this:

```
$ sudo yum info trustedanalytics-rest-server
```

Example results:

```
Available Packages
Name      : trustedanalytics-rest-server
Arch      : x86_64
Version   : #.#.#
Release   : #####
Size      : 419 M
Repo      : trustedanalytics
Summary   : trustedanalytics-rest-server-0.9
URL       : trustedanalytics.com
License   : Confidential
```

Troubleshooting Private Repository

- The most common errors when using the private repository:
 - Incorrect access token/key
 - Incorrect secret token/key
 - The server time is out of sync with the world
- Double check the access and secret keys in the ta.repo file.
- AWS S3 will fail with access denied errors if the system time is out of sync with the website. To keep the system time in sync with the website run:

```
$ sudo service ntpd start
```

- The Trusted Analytics Dependency repository and the yum-s3 package must be installed before the Trusted Analytics private repository.
- To use the yum command inside a corporate proxy make sure the `http_proxy` and `https_proxy` environment variables are set.
- The sudo command may need the `-E` option to maintain environment variables:

```
$ sudo -E yum command
```

16.3.2 Installing Trusted Analytics Package

Installing On The Master Node

Install Trusted Analytics Python REST server and its dependencies. Only one instance of the REST server needs to be installed. Installation location is flexible, but it is usually installed with the HDFS name node.

```
$ sudo yum -y install trustedanalytics-rest-server
```

Installing On A Worker Node

The Trusted Analytics spark dependencies package needs to be installed on every node running the spark worker role.

```
$ sudo yum -y install trustedanalytics-spark-deps
$ sudo yum -y install trustedanalytics-python-rest-client
```

16.4 REST Server Configuration

From the postgresql client, create a new database and user in postgresql. See the section on *postgresql*.

16.4.1 Configuration Script

The server configuration is semi-automated via the use of a Python script `/etc/trustedanalytics/rest-server/config.py`. It will query Cloudera Manager for the necessary configuration values and create a new `application.conf` file based on the `application.conf.tpl` file. The script will also fully configure the local PostgreSQL installation to work with the Trusted Analytics server.

To configure Trusted Analytics installation, do this:

```
$ cd /etc/trustedanalytics/rest-server/
$ sudo ./config
```

Answer the prompts to configure the cluster. To see an example of the prompts see [Configuration Script](#).

The script goes through all the necessary configurations to get the Trusted Analytics service running. The script can be run multiple times but there is a danger that configuring the database multiple times can wipe out a users data frames and graphs.

Command line arguments can also be supplied for every prompt. If a command line argument is given, no prompt will be presented. To get a list of all the command line arguments for the configuration script, run the same command with `--help`:

```
$ sudo ./config --help
```

16.4.2 Manual Configuration

This section is optional, but informative if additional changes to the configuration file are needed. (*Skip section*).

`/etc/trustedanalytics/rest-server/application.conf`

The REST server package provides a configuration template file which must be used to create a configuration file. Copy the configuration template file `application.conf.tpl` to `application.conf` in the same directory, like this:

```
$ cd /etc/trustedanalytics/rest-server
$ sudo cp application.conf.tpl application.conf
```

Open the file with a text editor:

```
$ sudo vi application.conf
```

All of the changes that need to be made are located at the top of the file. See [Appendix A — Sample Application Configuration File](#) for an example `application.conf` file.

Configure File System Root

Replace the text “invalid-fsroot-host” with the fully qualified domain of the HDFS Namenode.

Example:

```
fs.root = "hdfs://invalid-fsroot-host/user/atkuser"
```

Becomes:

```
fs.root = "hdfs://localhost.localdomain/user/atkuser"
```

If the HDFS Name Node port does not use the standard port, specify it after the host name with a colon:

```
fs.root = "hdfs://localhost.localdomain:8020/user/atkuser"
```

Configure Zookeeper Hosts

Replace the text “invalid-titan-host” with a comma delimited list of fully qualified domain names of all nodes running the zookeeper service.

Example:

```
titan.load.storage.hostname = "invalid-titan-host"
```

Becomes:

```
titan.load.storage.hostname = "localhost.localdomain,localhost.localdomain"
```

If the zookeeper client port is not 2181, un-comment the following line and replace 2181 with the zookeeper client port:

```
titan.load.storage.port = "2181"
```

Configure Spark Master Host

Update “invalid-spark-master”.

To run Spark on Yarn in yarn-cluster mode, set:

```
spark.master = yarn-cluster
```

To run Spark on Yarn in yarn-client mode, set:

```
spark.master = yarn-client
```

Configure Spark Executor Memory

The Spark executor memory needs to be set equal to or less than what is configured in Cloudera Manager. The Cloudera Spark installation will, by default, set the Spark executor memory to 8g, so 8g is usually a safe setting.

Example:

```
spark.executor.memory = "invalid executor memory"
```

Becomes:

```
spark.executor.memory = "8g"
```

Click on the Spark service then configuration in Cloudera Manager to get executor memory. See [Fig. 12.1](#).

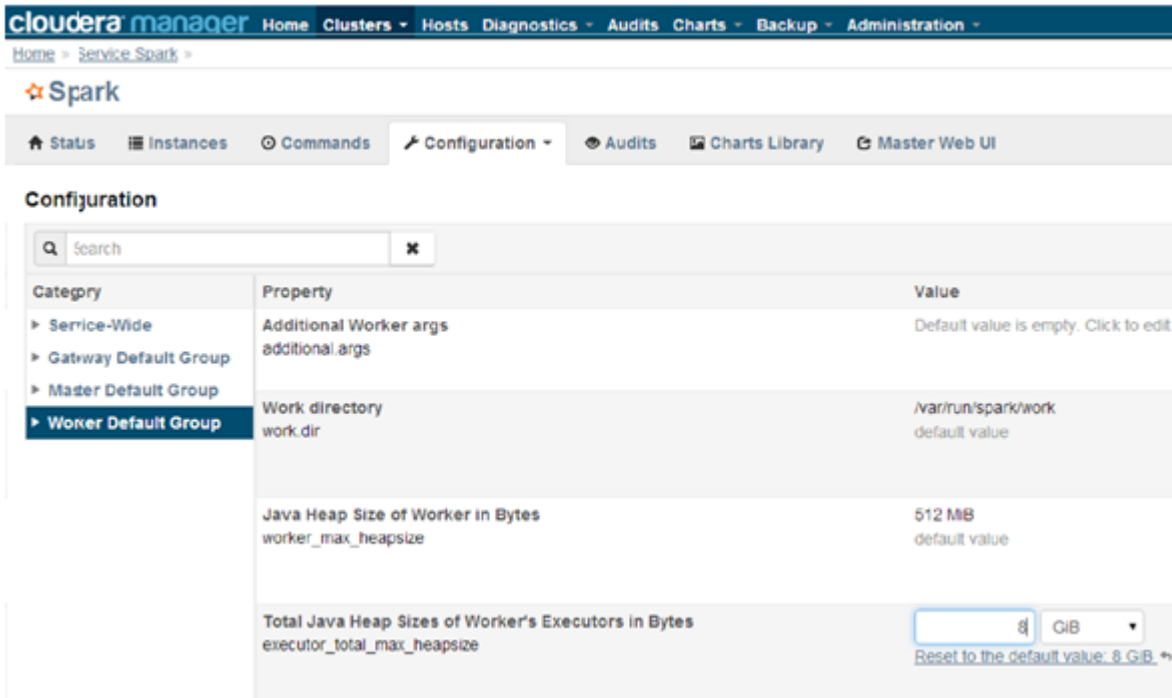


Fig. 16.1: Spark Executor Memory

Set the Bind IP Address (Optional)

The Trusted Analytics server can bind to all IP addresses, as opposed to just a single address, by updating the following lines and follow the commented instructions. This configuration section is also near the top of the file.

```
#bind address - change to 0.0.0.0 to listen on all interfaces
//host = "127.0.0.1"
```

Updating the Spark Class Path

The automatic configuration script updates the classpath in Cloudera Manager. The spark class path can also be configured through Cloudera Manager under the spark configuration / Worker Environment Advanced Configuration Snippet. See [Fig 12.2](#). If it isn't already set, add:

```
SPARK_CLASSPATH="/usr/lib/trustedanalytics/graphbuilder/lib/ispark-deps.jar"
```

End of manual configuration

Restart the Spark service. See [Fig. 13.3](#).

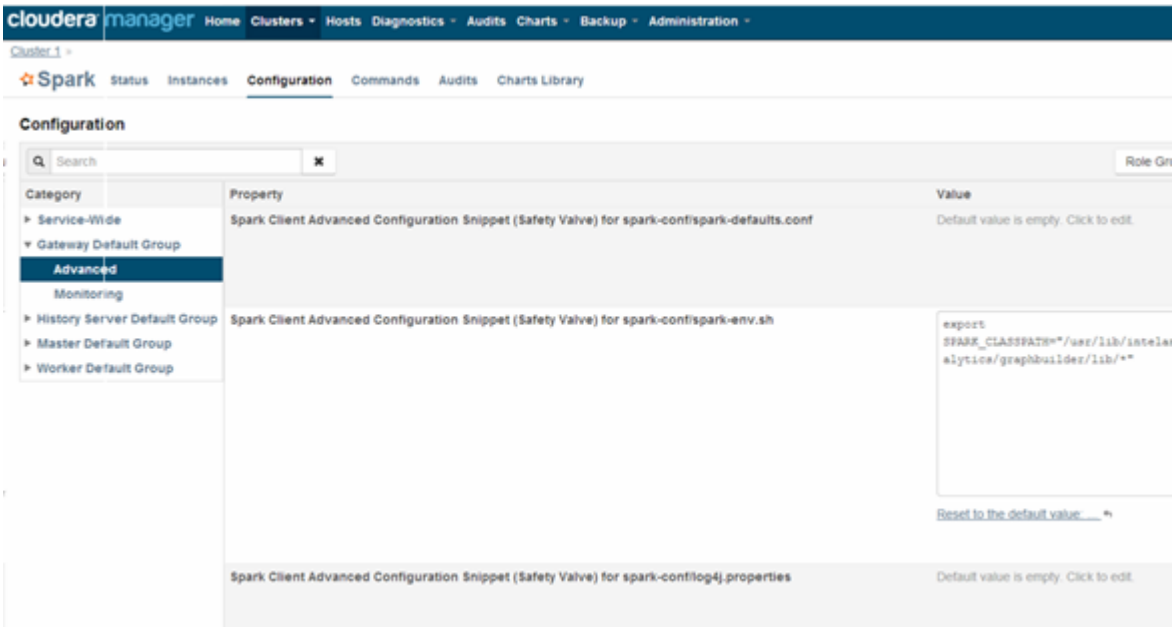


Fig. 16.2: Spark Class Path

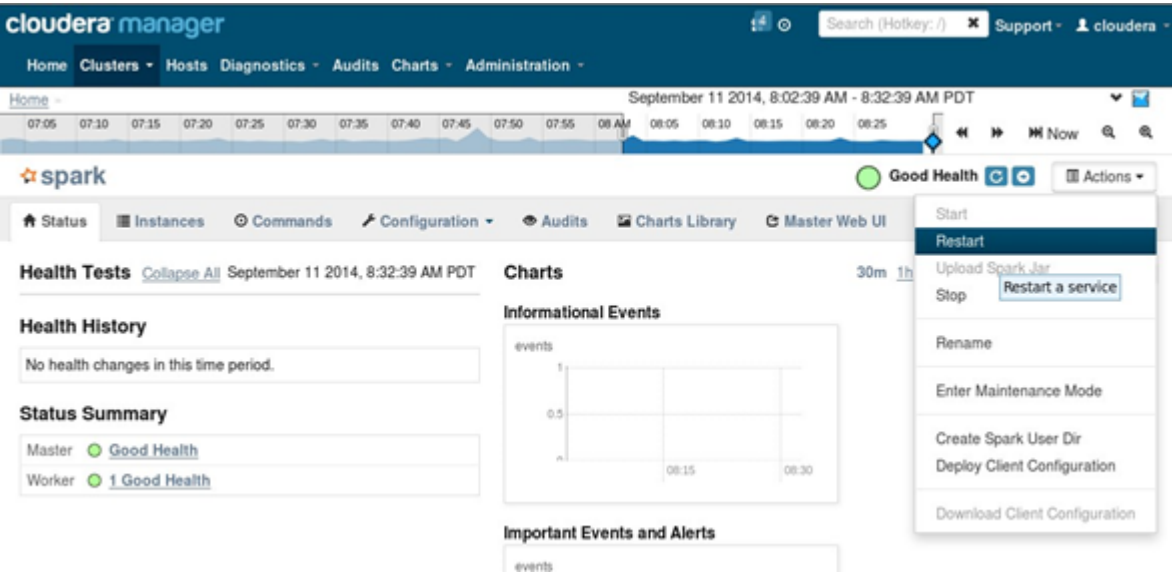


Fig. 16.3: Restart Spark

16.4.3 Database Configuration

The Trusted Analytics service can use two different databases H2 and PostgreSQL. The configuration script configures postgresql automatically.

H2

Caution: H2 will lose all metadata upon service restart.

Enabling H2 is very easy and only requires some changes to *application.conf*. To comment a line in the configuration file either prepend the line with two forward slashes `//` or a pound sign `#`.

The following lines need to be commented:

Before:

```
metastore.connection-postgresql.host = "invalid-postgresql-host"
metastore.connection-postgresql.port = 5432
metastore.connection-postgresql.database = "ta-metastore"
metastore.connection-postgresql.username = "atkuser"
metastore.connection-postgresql.password = "myPassword"
metastore.connection-postgresql.url = "jdbc:postgresql://" + ${trustedanalytics.atk.
    metastore.connection-postgresql.host} + ":" + ${trustedanalytics.atk.metastore.
    connection-postgresql.port} + "/" + ${trustedanalytics.atk.metastore.connection-
    postgresql.database}
metastore.connection = ${trustedanalytics.atk.metastore.connection-postgresql}
```

After:

```
//metastore.connection-postgresql.host = "invalid-postgresql-host"
//metastore.connection-postgresql.port = 5432
//metastore.connection-postgresql.database = "ta-metastore"
//metastore.connection-postgresql.username = "atkuser"
//metastore.connection-postgresql.password = "myPassword"
//metastore.connection-postgresql.url = "jdbc:postgresql://" + ${trustedanalytics.atk.
    metastore.connection-postgresql.host} + ":" + ${trustedanalytics.atk.metastore.
    connection-postgresql.port} + "/" + ${trustedanalytics.atk.metastore.connection-
    postgresql.database}
//metastore.connection = ${trustedanalytics.atk.metastore.connection-postgresql}
```

Next, uncomment the following line:

Before:

```
//metastore.connection = ${trustedanalytics.atk.metastore.connection-h2}
```

After:

```
metastore.connection = ${trustedanalytics.atk.metastore.connection-h2}
```

PostgreSQL

PostgreSQL configuration is more involved than H2 configuration and should only be attempted by an advanced user. Using PostgreSQL allows graphs and frames to persist across service restarts.

First, log into the postgres user on the linux system:

```
$ sudo su postgres
```

Start the postgres command line client:

```
$ psql
```

Wait for the command line prompt to come:

```
postgres=#
```

Then create a user:

```
postgres=# create user YOURUSER with createdb encrypted password 'YOUR_PASSWORD';
```

User creation confirmation:

```
CREATE ROLE
```

Then create a database for that user:

```
postgres=# create database YOURDATABASE with owner YOURUSER;
```

Database creation confirmation:

```
CREATE DATABASE
```

After creating the database exit the postgres command line by hitting `ctrl + d`

Once the database and user are created, open `/var/lib/pgsql/data/pg_hba.conf` and add this line `host all YOURUSER 127.0.0.1/32 md5` to very the top of the file:

```
$ vi /var/lib/pgsql/data/pg_hba.conf
```

Add the new line at the very top of the file or before any uncommented lines. If the `pg_hba.conf` file doesn't exist, initialize postgresql with:

```
$ sudo survice postgresql initdb
```

Now that the database is created, uncomment all the postgres lines in `application.conf`.

Before:

```
//metastore.connection-postgresql.host = "invalid-postgresql-host"
//metastore.connection-postgresql.port = 5432
//metastore.connection-postgresql.database = "ta-metastore"
//metastore.connection-postgresql.username = "atkuser"
//metastore.connection-postgresql.password = "myPassword"
//metastore.connection-postgresql.url = "jdbc:postgresql://"
    ${trustedanalytics.atk.metastore.connection-postgresql.host}":"
    ${trustedanalytics.atk.metastore.connection-postgresql.port}"/"
    ${trustedanalytics.atk.metastore.connection-postgresql.database}
//metastore.connection = ${trustedanalytics.atk.metastore.connection-postgresql}
```

After:

```
metastore.connection-postgresql.host = "localhost"
metastore.connection-postgresql.port = 5432
metastore.connection-postgresql.database = "YOURDATABASE"
metastore.connection-postgresql.username = "YOURUSER"
metastore.connection-postgresql.password = "YOUR_PASSWORD"
metastore.connection-postgresql.url = "jdbc:postgresql://"
    ${trustedanalytics.atk.metastore.connection-postgresql.host}":"
```

```

    ${trustedanalytics.atk.metastore.connection-postgresql.port}"/"
    ${trustedanalytics.atk.metastore.connection-postgresql.database}
metastore.connection = ${trustedanalytics.atk.metastore.connection-postgresql}
#comment any h2 configuration lines with a # or //::
//metastore.connection = ${trustedanalytics.atk.metastore.connection-h2}

```

Restart the Trusted Analytics service:

```
$ sudo service trustedanalytics restart
```

After restarting the service, Trusted Analytics will create all the database tables. Now insert a meta user to enable Python client requests.

Login to the postgres linux user:

```
$ sudo su postgres
```

Open the postgres command line:

```
$ psql
```

Switch databases:

```

postgres=# \c YOURDATABASE
psql (8.4.18)
You are now connected to database "YOURDATABASE".

```

Then insert into the users table:

```

postgres=# insert into users (username, api_key, created_on,
        modified_on) values( 'metastore', 'test_api_key_1', now(),
        now() );
INSERT 0 1

```

View the insertion by doing a select on the users table:

```
postgres=# select * from users;
```

There should only be a single row per api_key:

user_id	username	api_key	created_on	modified_on
1	metastore	api_key_1	2014-11-20 12:37:16	2014-11-20 12:37:16

(1 row)

If there is more than one row for a single api key, remove one of them or create a new database. The server will not be able to validate a request from the REST client if there are duplicate api keys.

After the confirmation of the insert, commands from the python client can be sent.

16.4.4 Starting The Trusted Analytics REST Server

Starting the REST server is very easy. It can be started like any other Linux service.

```
$ sudo service trustedanalytics start
```

After starting the REST server, browse to the host on port 9099 (<master node ip address>:9099) to see if the server started successfully.

16.4.5 Troubleshooting Trusted Analytics REST Server

A log gets written to `/var/log/trustedanalytics/rest-server/output.log` or `/var/log/trustedanalytics/rest-server/application.log`. To resolve issues starting or running jobs, tail either log to see what error is getting reported while running the task:

```
$ sudo tail -f /var/log/trustedanalytics/rest-server/output.log
```

or:

```
$ sudo tail -f /var/log/trustedanalytics/rest-server/application.log
```

16.4.6 Upgrading

Unless specified otherwise in the release notes, upgrading requires removal of old software prior to installation of new software.

Configuration Script

Sample output with notes:

```
#if the default is correct hit enter
$ sudo ./config

What port is Cloudera Manager listening on?
  defaults to "7180" if nothing is entered:
What is the Cloudera Manager username?
  defaults to "admin" if nothing is entered:
What is the Cloudera Manager password?
  defaults to "admin" if nothing is entered:

No current SPARK_CLASSPATH set.
Setting to:

Deploying config . . . . .
Config Deployed

You need to restart Spark service for the config changes to take affect.
Would you like to restart now? Enter 'yes' to restart. defaults to 'no' if nothing is
entered: yes
Restarting Spark . . . . .
Restarted Spark

What is the hostname of the database server?
  defaults to "localhost" if nothing is entered:
What is the port of the database server?
  defaults to "5432" if nothing is entered:
What is the name of the database?
  defaults to "ta_metastore" if nothing is entered:
What is the database user name?
  defaults to "atkuser" if nothing is entered:

#The dollar sign($) is not allowed in the password.
What is the database password?
  The default password was randomly generated.
```

```

Defaults to "*****" if nothing is entered.

Creating application.conf file from application.conf.tpl
Reading application.conf.tpl
Updating configuration
Configuration created for Trusted Analytics ATK
Configuring postgres access for "atkuser"
Initializing database:                [ OK ]
Stopping postgresql service:          [ OK ]
Starting postgresql service:          [ OK ]
0
CREATE ROLE
0
CREATE DATABASE
0
Stopping postgresql service:          [ OK ]
Starting postgresql service:          [ OK ]
0
initctl: Unknown instance:
trustedanalytics-rest-server start/running, process 17484
0
Waiting for Trusted Analytics ATK server to restart
. . . .
You are now connected to database "ta_metastore".
INSERT 0 1
0
postgres is configured
Trusted Analytics ATK is ready for use.

```


SCORING ENGINE

This section covers the scoring engine installation, configuration and running the scoring engine.

17.1 Installation

The scoring engine repositories are automatically installed as part of the ATK repositories. Please see section ATK packages installation.

17.2 Scoring Models Implementation

The scoring engine is a generic engine and is oblivious to the streaming scoring model. It is up to the user of the engine to provide the scoring implementation. The implementation, model bytes and the class name is provided in a tar file at the startup of the scoring engine. The scoring engine expects three files in the tar file as listed below.

1. The model implementation jar file. This is the
2. A file by the name `modelname.txt` that contains the name of the class that implements the scoring in the jar file.
3. The file that will have the model bytes that will be used by the scoring. The name of this file will be the name of the URL of the rest server. See section [starting_scoring_engine](#) below.

Note: If you are using ATK to score, the publish method on the models will create the tar file that can be used as input to the scoring engine.

17.3 Configuration of the Engine

/etc/trustedanalytics/scoring/application.conf

The scoring engine provides a configuration template file which must be used to create a configuration file. Copy the configuration template file *application.conf.tpl* in the same directory, like this:

```
$ cd /etc/trustedanalytics/scoring
$ sudo cp application.conf.tpl application.conf
```

Open the file with a text editor:

```
$ sudo vi application.conf
```

Modify the section below to point to the where the scoring tar file is located. Below is an example:

```
trustedanalytics.scoring-engine {  
    archive-tar = "hdfs://scoring-server.intel.com:8020/user/atkuser/kmeans.tar"  
}
```

17.4 Starting the Scoring Engine Service

Once the application.conf file has been modified to point to the scoring tar file, the scoring engine can be started with the following command:

```
$ sudo service scoring-engine start
```

This will launch the rest server for the engine. The REST API is:

```
GET /v1/models/[name]?data=[urlencoded record 1]
```

17.5 Scoring Client

Below is a sample python script to connect to the scoring engine:

```
>>> import requests  
>>> import json  
>>> headers = {'Content-type': 'application/json',  
...           'Accept': 'application/json,text/plain'}  
>>> r = requests.post('http://localhost:9099/v1/models/testjson?data=2', headers=headers)
```


VIRTUAL MACHINES

Table of Contents

- *Introduction*
- *Requirements*
- *Download VM (Virtual Machine) Image*
- *Windows GUI Client*
- *Extract Archive*
 - *Extracting On Windows*
 - *Extracting On Linux*
- *Import Image*
- *Running Trusted Analytics VM Image*
 - *Before starting*
 - *Sample Scripts*
- *Eclipse/PyDev*
- *Logs*
- *Updating*
- *Common VM problems*

18.1 Introduction

This guide goes through the download and import of Trusted Analytics beta on a virtual machine (VM). Currently Trusted Analytics VM only supports [Virtual Box](https://www.virtualbox.org/)¹. These instructions do not cover the installation of Virtual Box. Virtual Box supports many platforms and can be [downloaded for free](https://www.virtualbox.org/wiki/Downloads)². The installation documentation is also [available online](https://www.virtualbox.org/manual/UserManual.html)³.

18.2 Requirements

- 12GB of memory needs to be allocated to the VM
- 45GB of free hard drive space
- Working Virtual Box 4.3 installation

¹<https://www.virtualbox.org/>

²<https://www.virtualbox.org/wiki/Downloads>

³<https://www.virtualbox.org/manual/UserManual.html>

18.3 Download VM Image

Open a Linux shell (or for Windows user a command prompt) to run the various commands.

The VM image is downloaded from AWS. The download requires the AWS Command Line Interface (CLI) client. Instructions for downloading and installing CLI can be found at [Amazon cli documentation](#)⁴.

After installing the interface, verify the installation by running:

```
$ aws --version
```

The result should be similar to this:

```
aws-cli/1.2.9 Python/2.7.6 Linux/3.8.0-35-generic
```

Take note of the aws-cli version, as it must be greater or equal to 1.2.9. Older versions of the aws-cli client does not work with the restricted permissions.

The aws-cli client can be updated with pip or download the new Windows MSI (reference [Windows GUI Client](#)).

```
$ sudo pip install -U awscli
```

After aws installation, run:

```
$ aws configure
```

The program prompts for the access and secret tokens given at registration. When prompted for the “Default region name”, use “us-west-2”. When prompted for the “Default output format”, use “json”.

```
AWS Access Key ID [None]: <my access key>
AWS Secret Access Key [None]: <my secret key>
Default region name [None]: <us-west-2>
Default output format [None]: <json>
```

List the files in the directory:

```
$ aws s3 ls s3://trustedanalytics-repo/release/latest/vm/
2014-08-19 12:57:03          0
2014-11-25 16:22:57          70 TrustedAnalytics-VM.md5
2014-11-25 16:22:57 14656025025 TrustedAnalytics-VM.tar.gz
```

Download the tar.gz file. In this case, it’s ‘TrustedAnalytics-VM.tar.gz’:

```
$ aws s3 cp s3://trustedanalytics-repo/release/latest/vm/TrustedAnalytics-VM.tar.gz ./
```

[Skip section about Windows GUI Client].

18.4 Windows GUI Client

If you are on a Windows machine, and you prefer a GUI client, use the [S3Browser](#)⁵ to download the VM.

1. Download the Windows MSI <http://s3browser.com/download.php>.
2. Install and open the S3Browser application.
3. Add the keys provided.

⁴<http://docs.aws.amazon.com/cli/latest/userguide/installing.html>

⁵<http://s3browser.com/>

- (a) Navigate to:
 - i. **Accounts**
 - ii. **Add new account**
 or press **Ctrl + Shift + A**.

See *Fig. 15.1*.

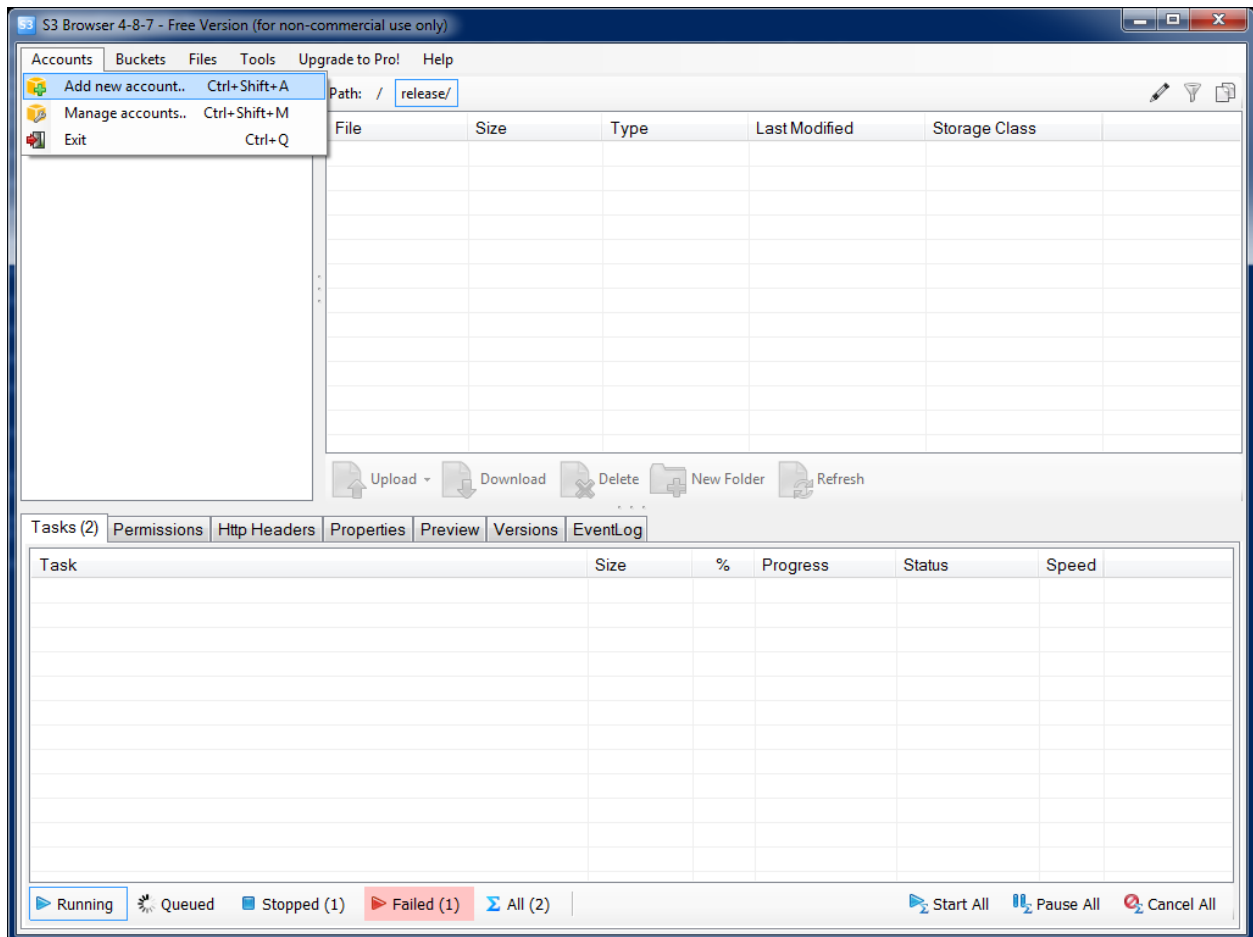


Fig. 18.1: Add New Account

- (b) In the account creation window:
 - i. Add your access and secret keys
 - ii. Give the account a name

See *Fig. 15.2*.

- 4. Navigate to:

- (a) **Buckets**
- (b) **Add External Bucket**

or press **Ctrl + E**.

See *Fig. 15.3*.

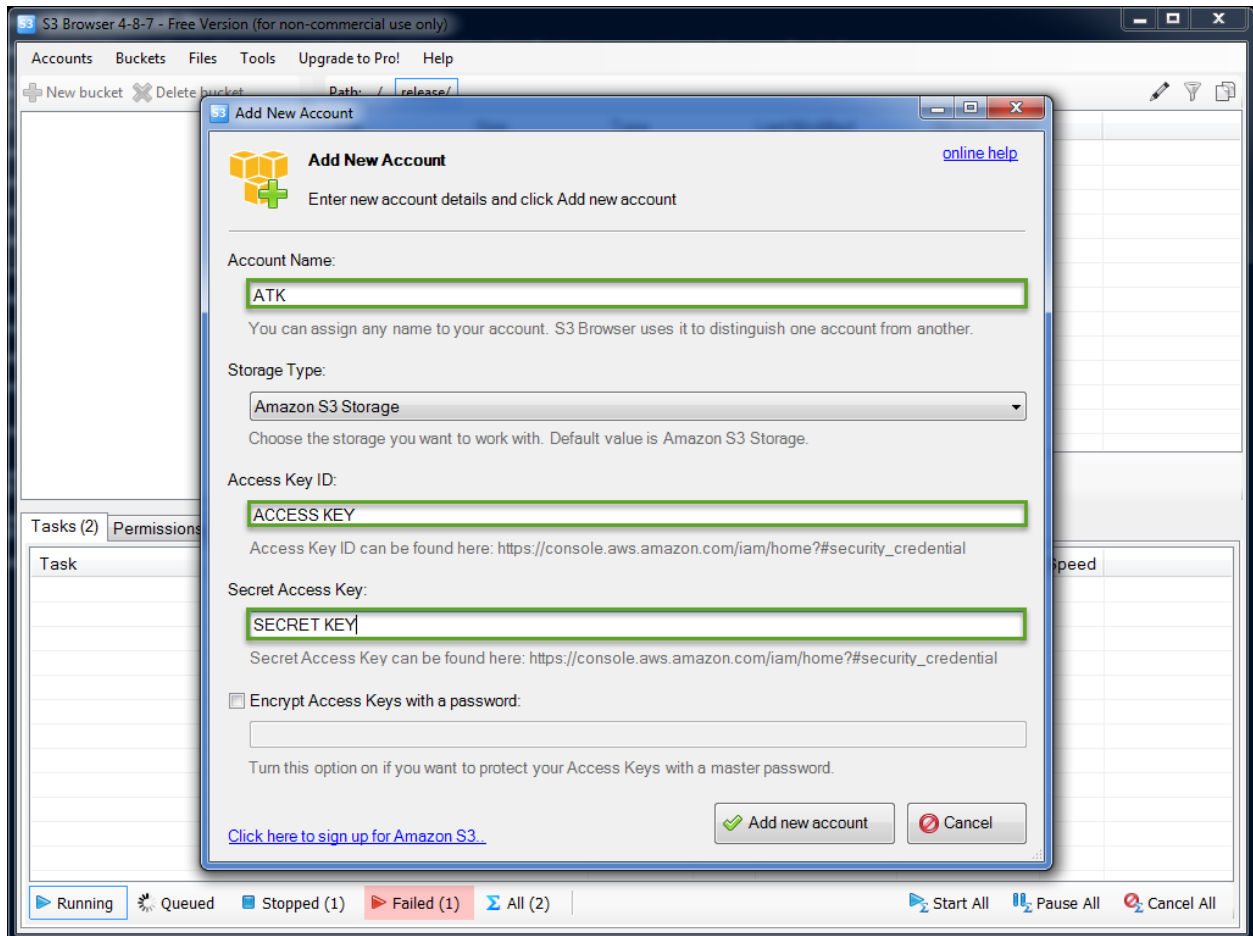


Fig. 18.2: New Account Information

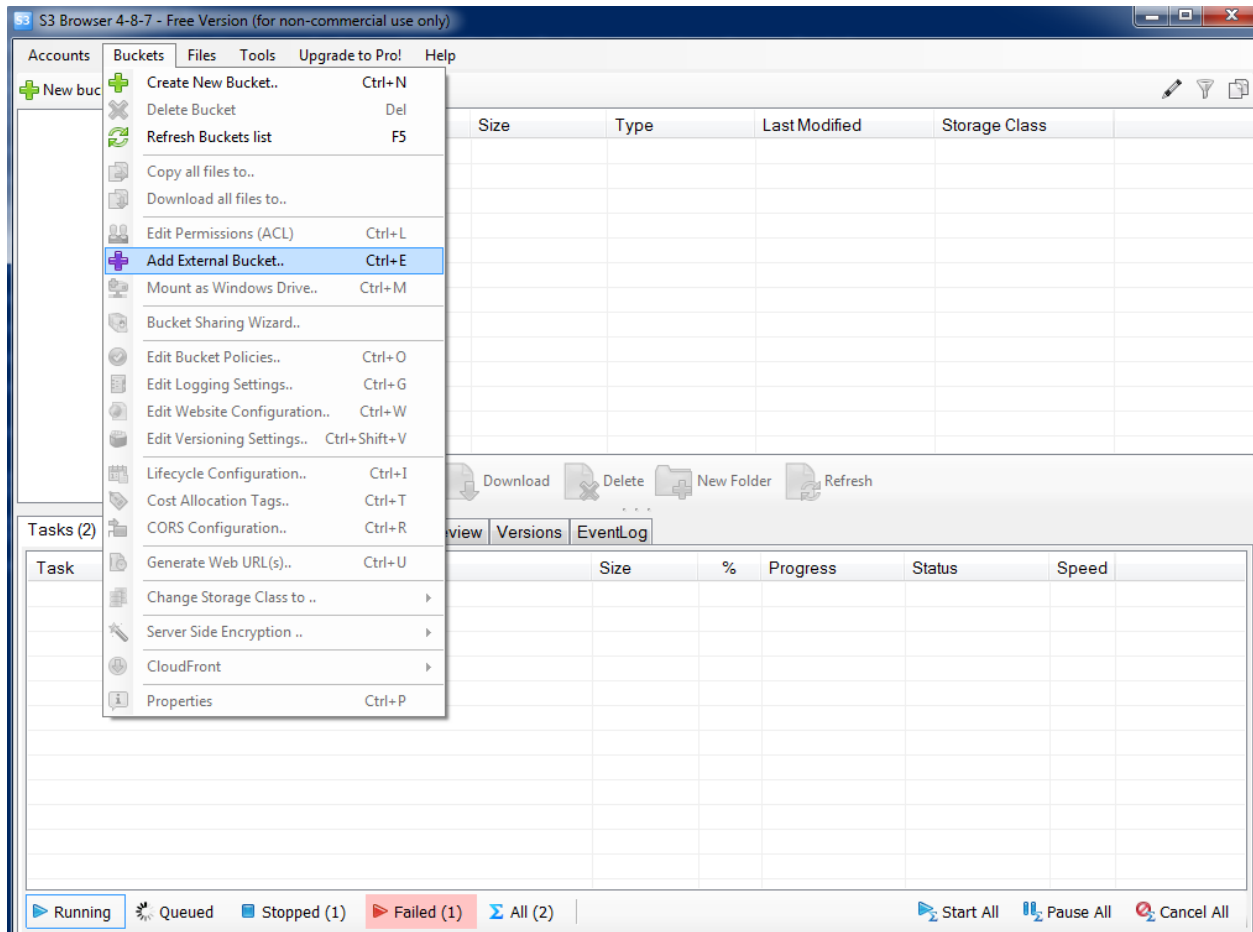


Fig. 18.3: Add External Bucket

5. Add the bucket url “trustedanalytics-repo/release”, then click **Add External bucket**. See *Fig. 15.4*.

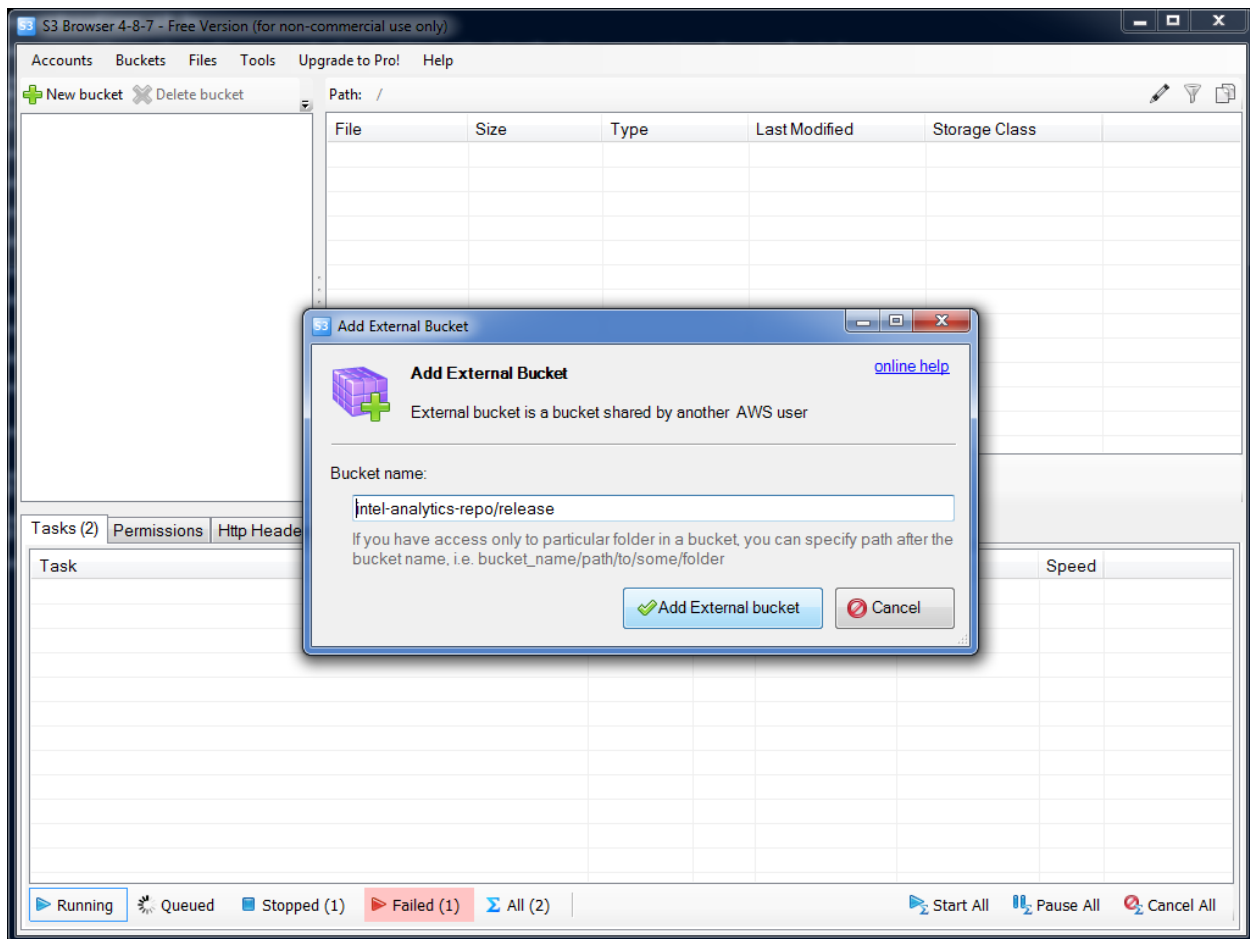


Fig. 18.4: Give Bucket Name

6. After adding the bucket, a list of folders shows up on the right. See *Fig. 15.5*.
7. Select the appropriate version, and navigate to the VM folder, then right click and download the “tar.gz” file. See *Fig. 15.6*.

18.5 Extract Archive

18.5.1 Extracting On Windows

Extracting on Windows is relatively easy. Use *7zip*⁶ (or equivalent tool) to extract the archive.

18.5.2 Extracting On Linux

After acquiring the VM, extract the archive:

⁶<http://7-zip.org/>

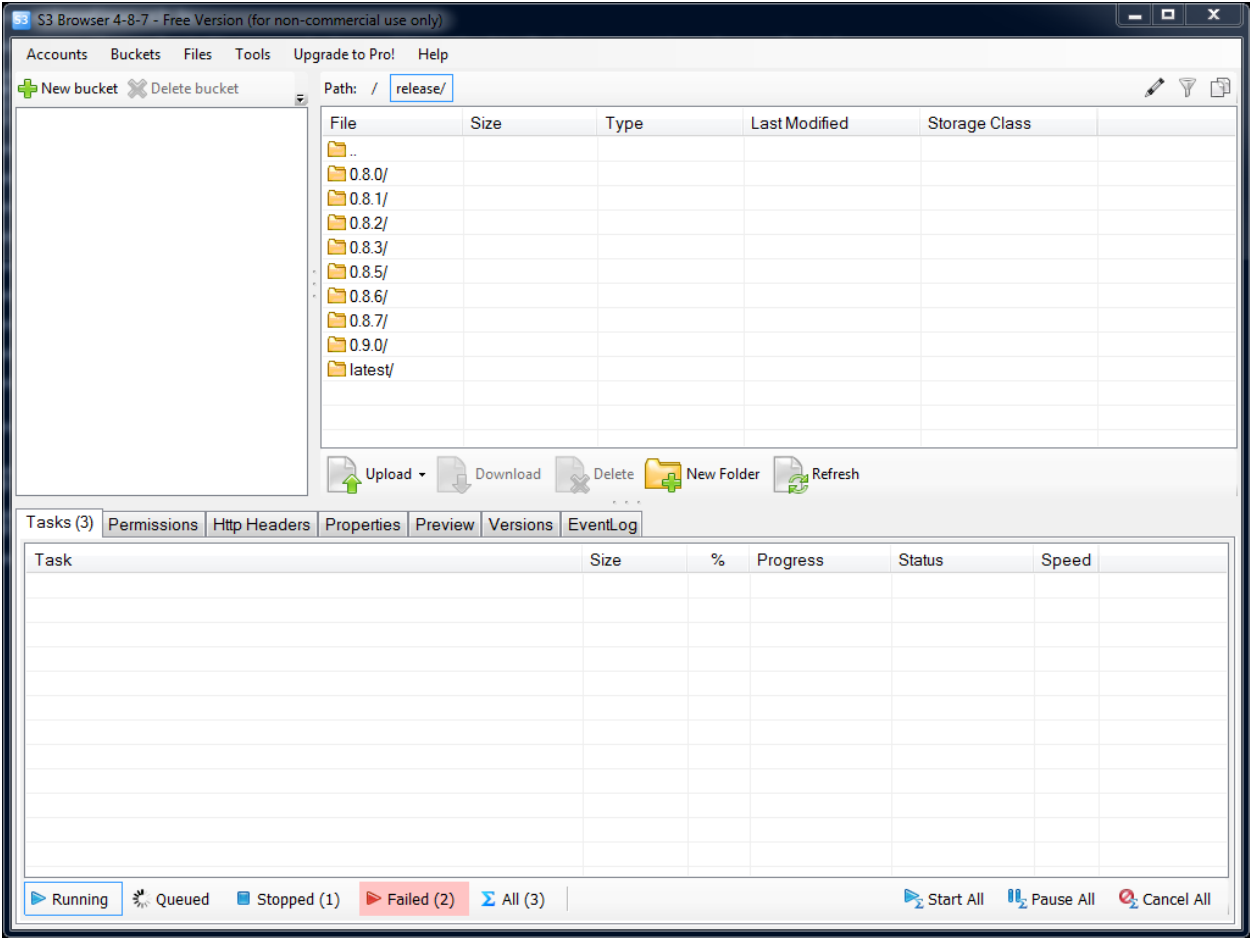


Fig. 18.5: Check Folder List

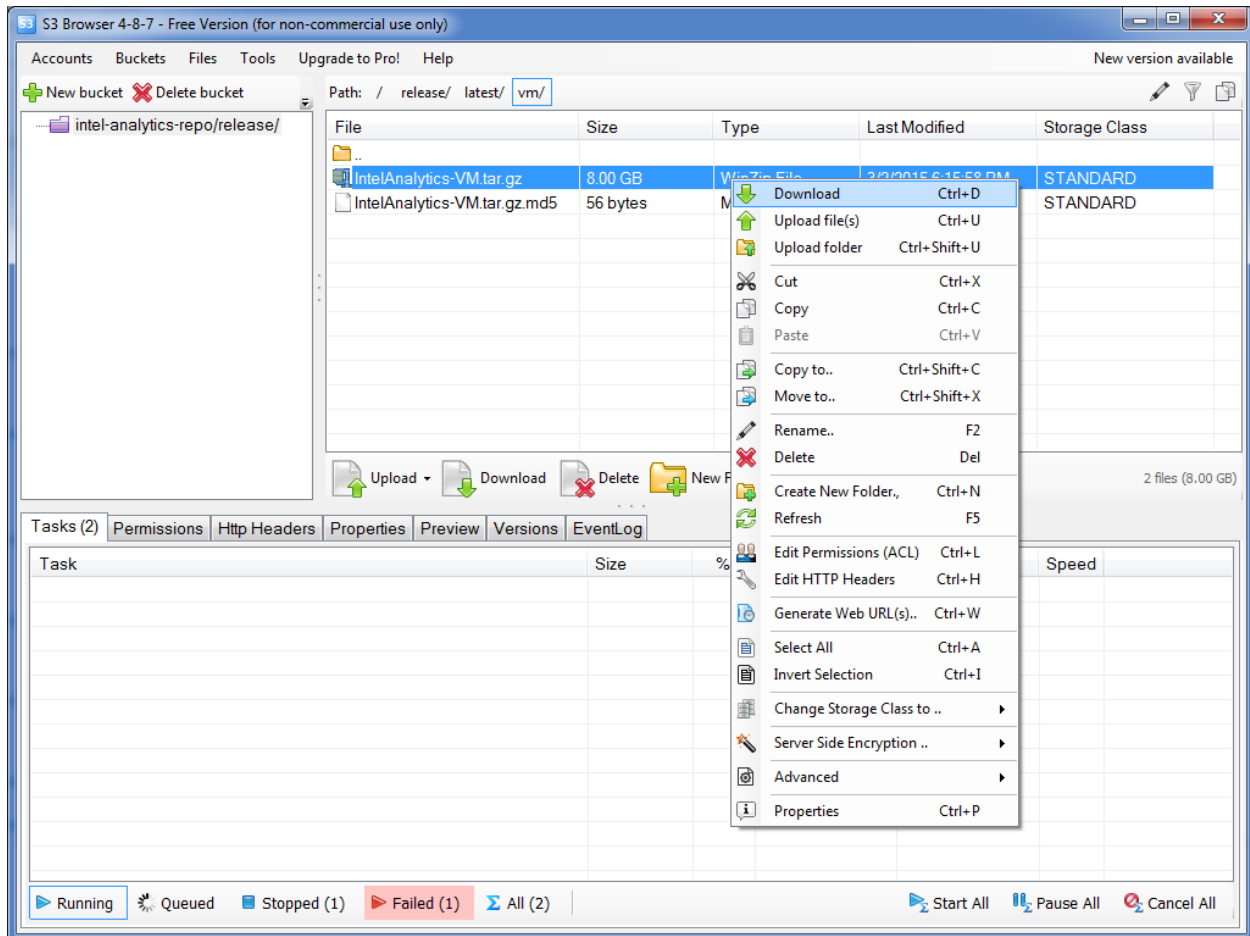


Fig. 18.6: Download File


```
$ tar -xvf TrustedAnalytics-VM.tar.gz
```

After extraction, there should be two (2) files, one with the extension ‘vmdk’, and another with the extension ‘ovf’.

18.6 Import Image

To import the VM image, do the following steps in Virtual Box.

1. Go to the **File** menu, then **Import Appliance**. See [Fig. 15.7](#).

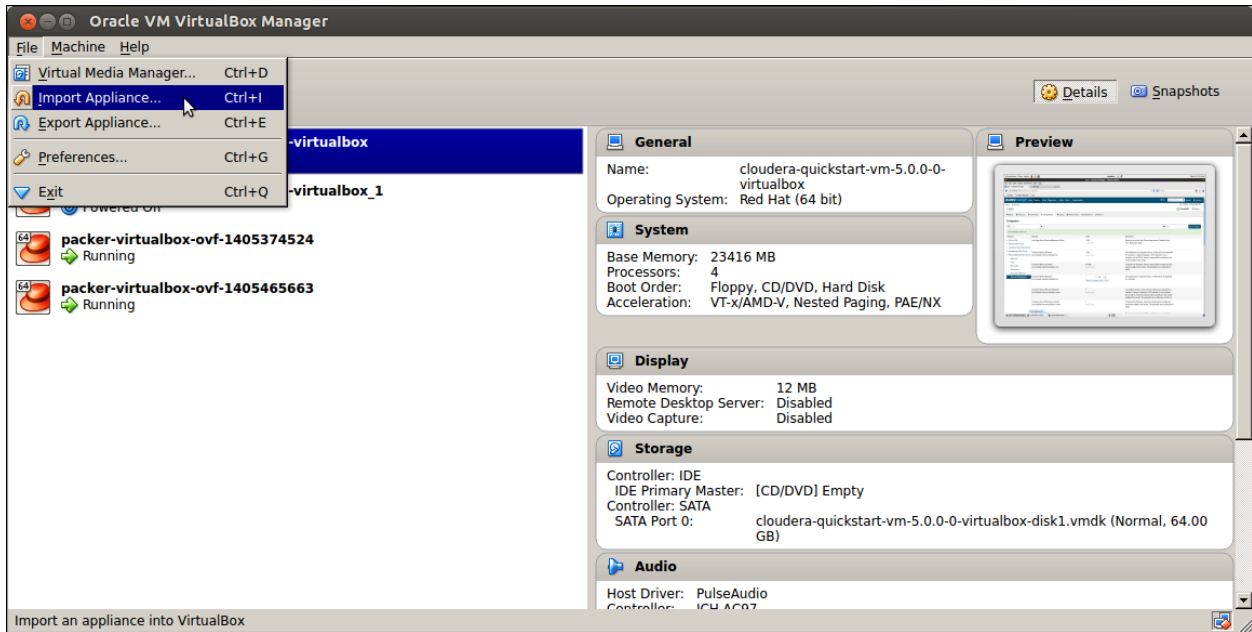


Fig. 18.7: File -> Import Appliance

2. Select the file with the extension ‘ovf’, which was extracted earlier from the VM image. See [Fig. 15.8](#).
3. Import the Trusted Analytics VM. See [Fig. 15.9](#).
4. After clicking **Import**, wait for the VM to be imported. See [Fig. 15.10](#).
5. Once the VM is imported, boot the VM by selecting the VM and clicking **Start**. See [Fig. 15.11](#).

18.7 Running Trusted Analytics VM Image

18.7.1 Before starting

After every reboot of the VM, the Trusted Analytics server must also be restarted.

```
$ sudo service trustedanalytics restart
```

Upon restart, if the service wasn’t running before it was told to stop, the system reports:

```
initctl: Unknown instance:
```

This message can be safely ignored.

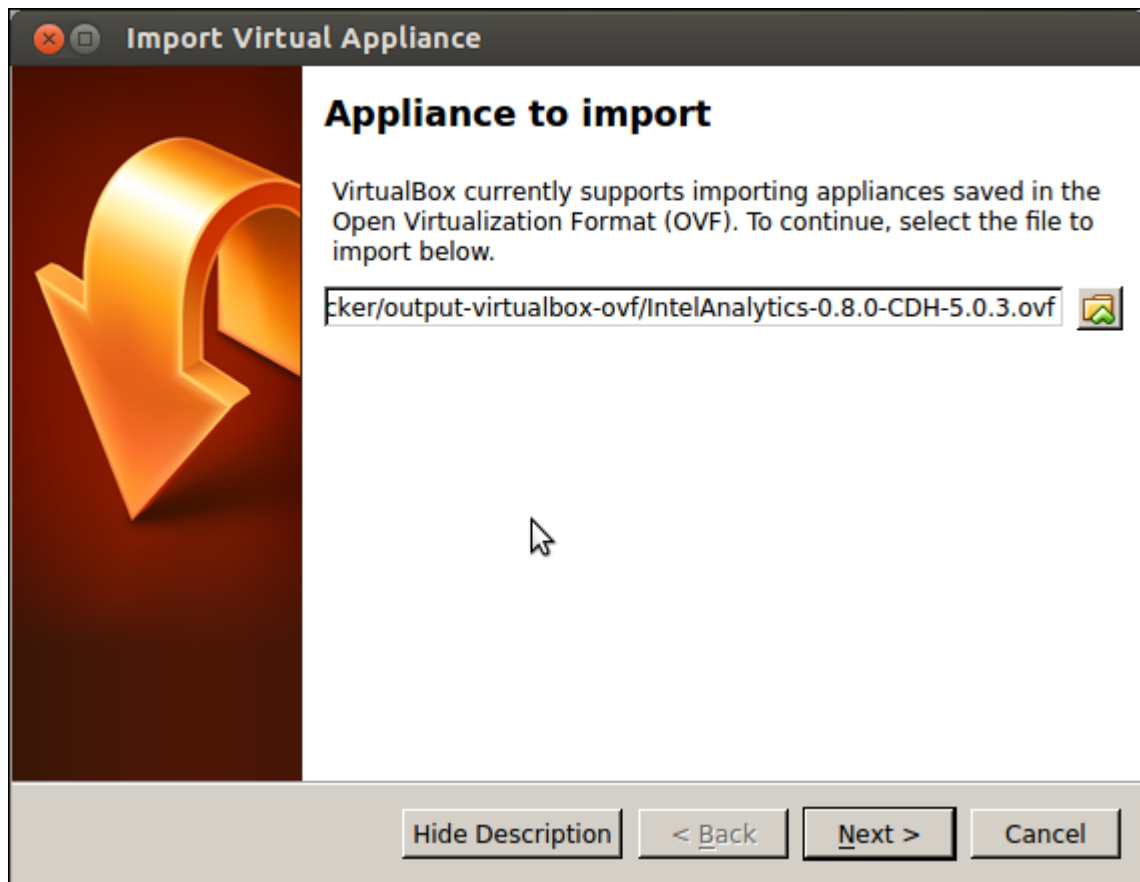


Fig. 18.8: Appliance to Import

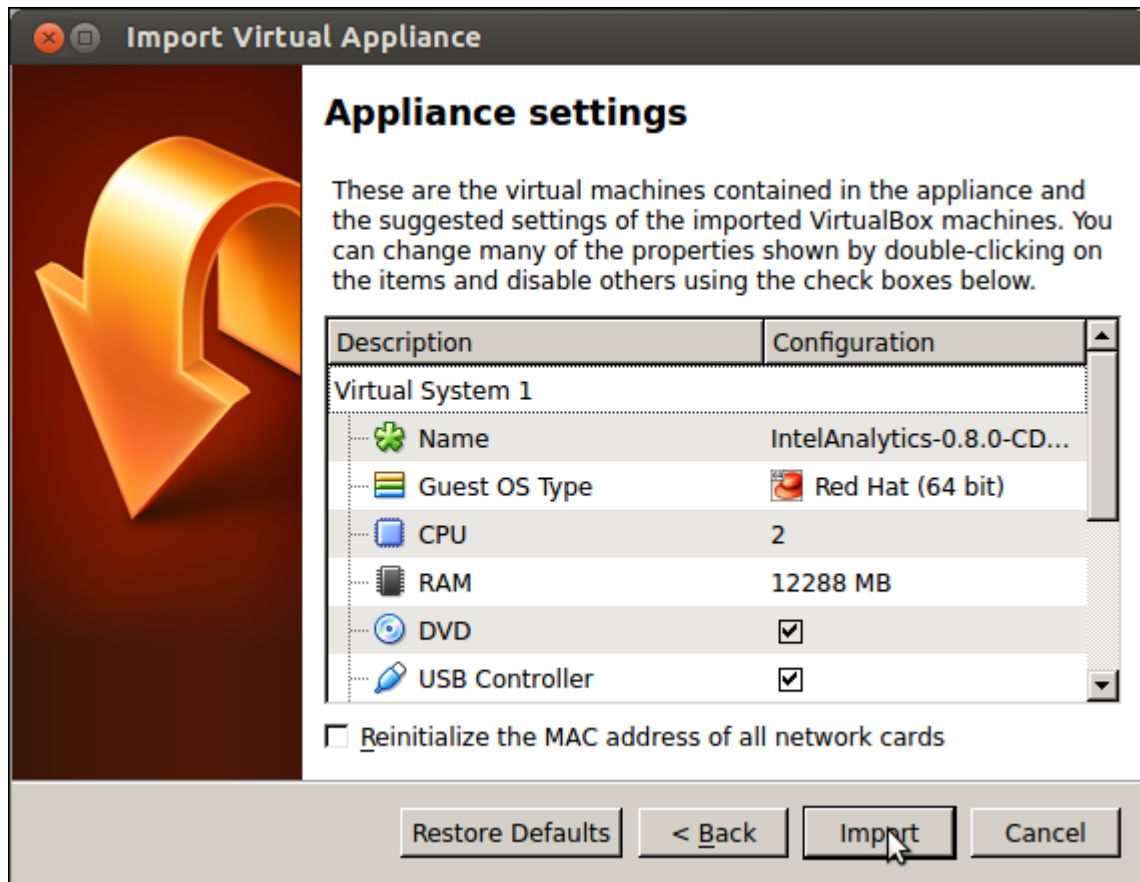


Fig. 18.9: Appliance Settings

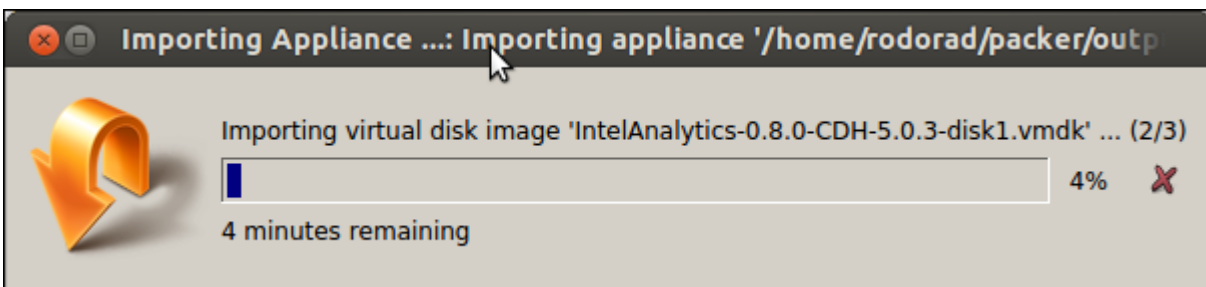


Fig. 18.10: Watching Appliance Import

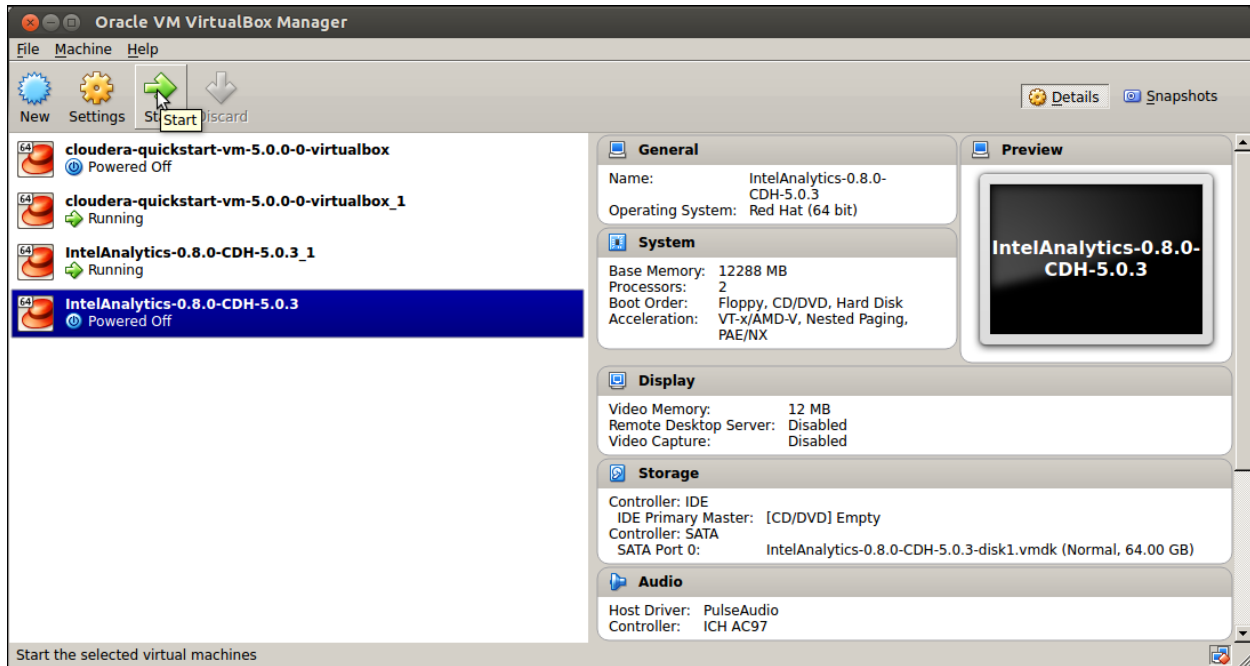


Fig. 18.11: Boot the VM

18.7.2 Sample Scripts

The VM is pre-configured and installed with Trusted Analytics. Several examples and datasets are included to get people familiar with the coding and behavior of Trusted Analytics.

The examples are located in '/home/cloudera/examples'.

```
drwxr-xr-x 2 cloudera cloudera 4096 Aug  1 00:53 datasets
-rw-r--r-- 1 cloudera cloudera 1100 Aug  1 10:15 lbp.py
-rw-r--r-- 1 cloudera cloudera  707 Aug  1 00:53 lda.py
-rw-r--r-- 1 cloudera cloudera  930 Aug  1 00:53 lp.py
```

The datasets are located in '/home/cloudera/examples/datasets' and 'hdfs://user/trustedanalytics/datasets/'.

```
-rw-r--r-- 1 atkuser atkuser      122 2014-08-01 /user/trustedanalytics/datasets/README
-rw-r--r-- 1 atkuser atkuser  617816 2014-08-01 /user/trustedanalytics/datasets/apl.csv
-rw-r--r-- 1 atkuser atkuser  8162836 2014-08-01 /user/trustedanalytics/datasets/lbp_edge.csv
-rw-r--r-- 1 atkuser atkuser  188470 2014-08-01 /user/trustedanalytics/datasets/lp_edge.csv
-rw-r--r-- 1 atkuser atkuser 311641390 2014-08-01 /user/trustedanalytics/datasets/test_lda.csv
```

The datasets in '/home/cloudera/examples/datasets' are for reference. The actual data that is being used by the Python examples and the Trusted Analytics server is in 'hdfs://user/trustedanalytics/datasets'.

To run any of the Python example scripts, start in the examples directory and start Python with the script name:

```
$ python <SCRIPT_NAME>.py
```

where <SCRIPT_NAME> is any of the scripts in '/home/cloudera/example'.

Example:

```
$ cd /home/cloudera/examples
$ python pr.py
```

18.8 Eclipse/PyDev

The VM comes with Eclipse and PyDev installed and ready for use. Importing the example scripts is easy.

1. Go to the desktop, and double-click on the Eclipse icon.
2. Go to **File** menu, and select **New** and then **Other**.

See [Fig. 15.12](#).

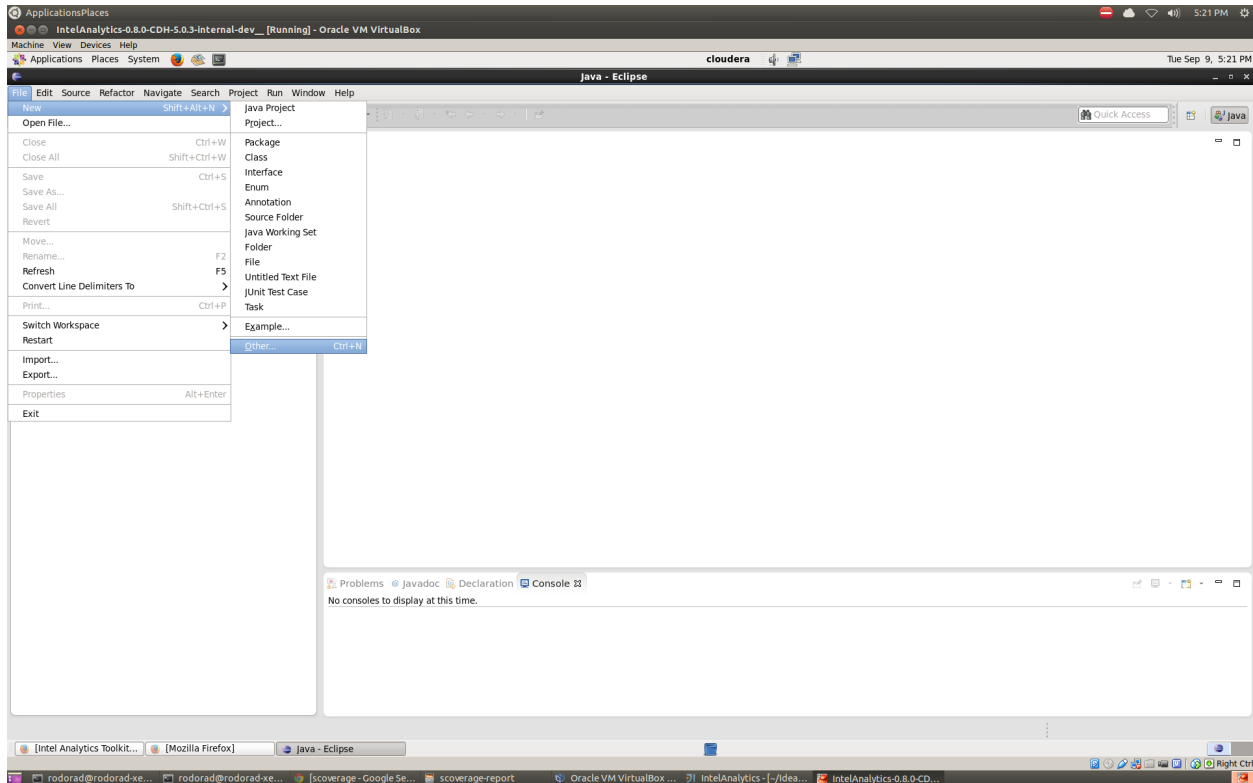


Fig. 18.12: Starting Eclipse

3. After selecting **File->New->Other**, look for the PyDev folder and expand the list, then select **PyDev Project** then click **Next**. See [Fig. 15.13](#).
4. The only field you have to change is the 'Project Contents' default directory. Uncheck 'Use default' and enter the directory you want to use '/home/cloudera/examples'. Everything else can be left with the default values. Click **Next** when you are done. See [Fig. 15.14](#).
5. You should now be able to see all the example scripts on the left hand pane. See [Fig. 15.15](#).

18.9 Logs

To debug changes to the scripts (or to peek behind the curtain), the log file is '/var/log/trustedanalytics/rest-server/output.log'. To show the log as it is generated, run `tail -f`:

```
$ sudo tail -f /var/log/trustedanalytics/rest-server/output.log
```

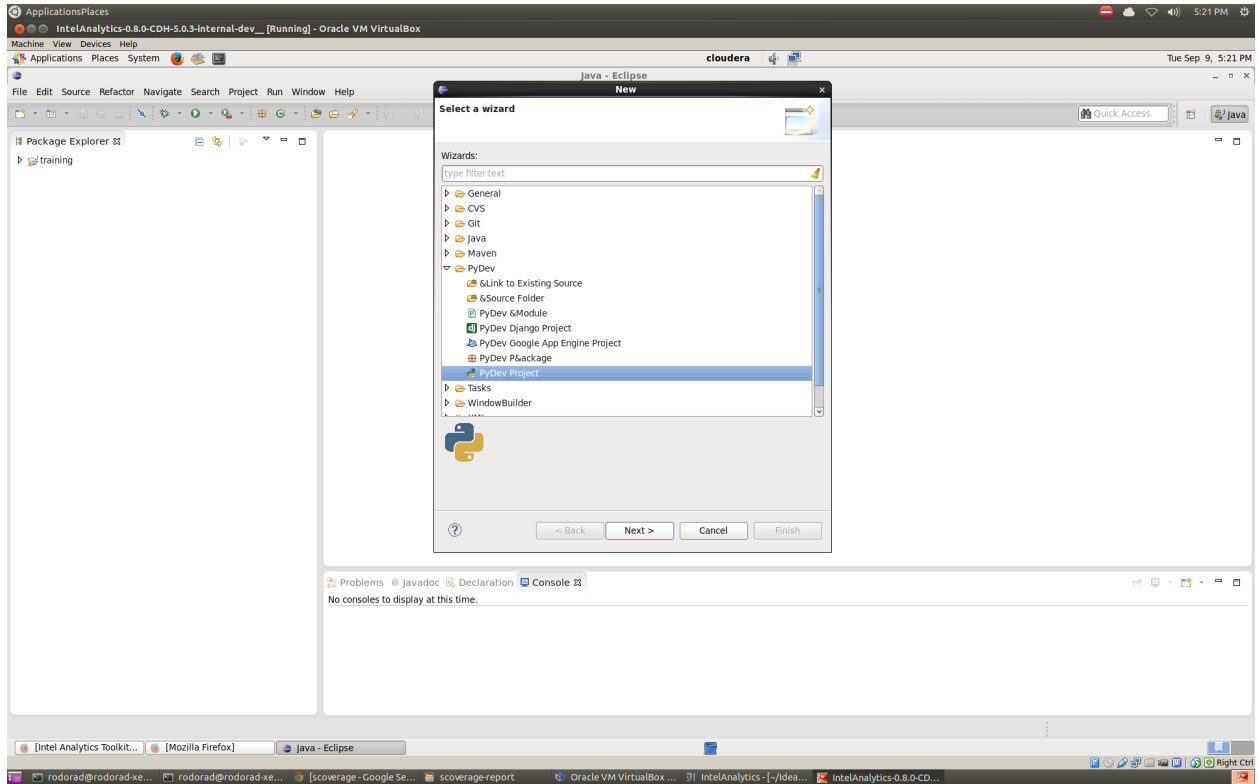


Fig. 18.13: New PyDev Project

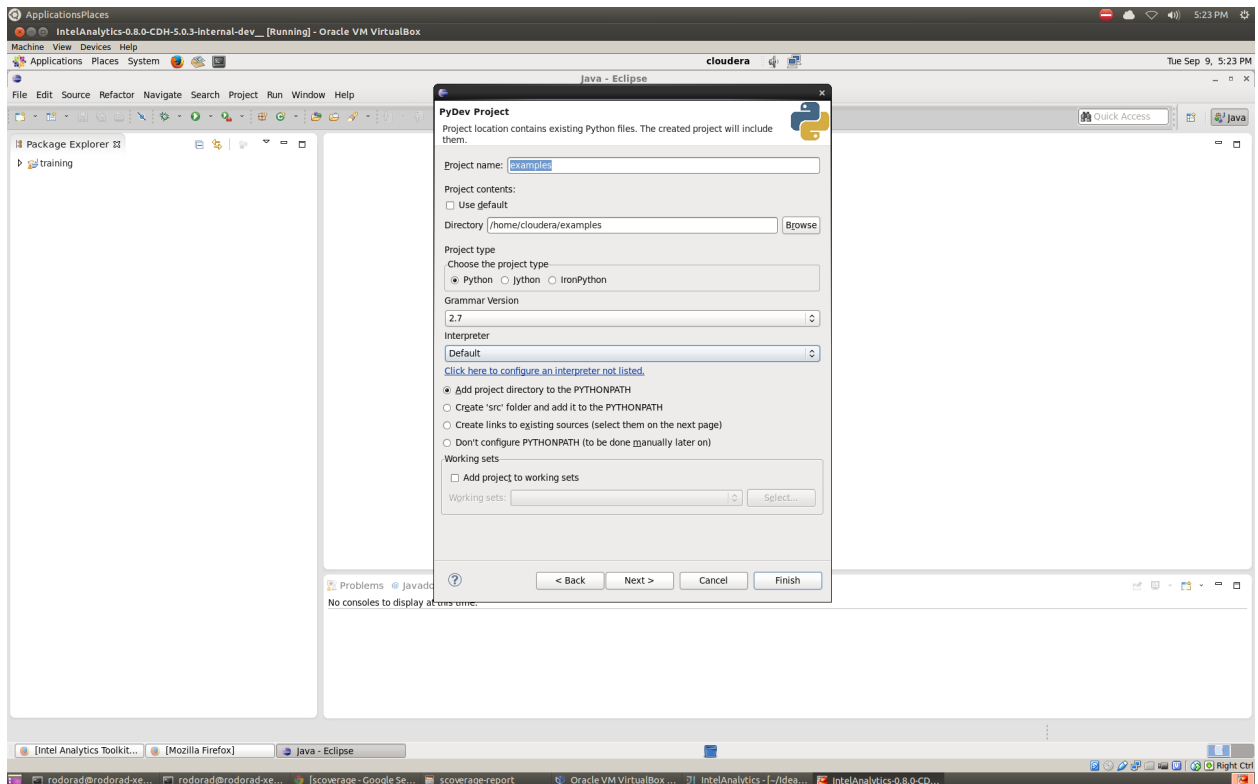


Fig. 18.14: Enter Working Path

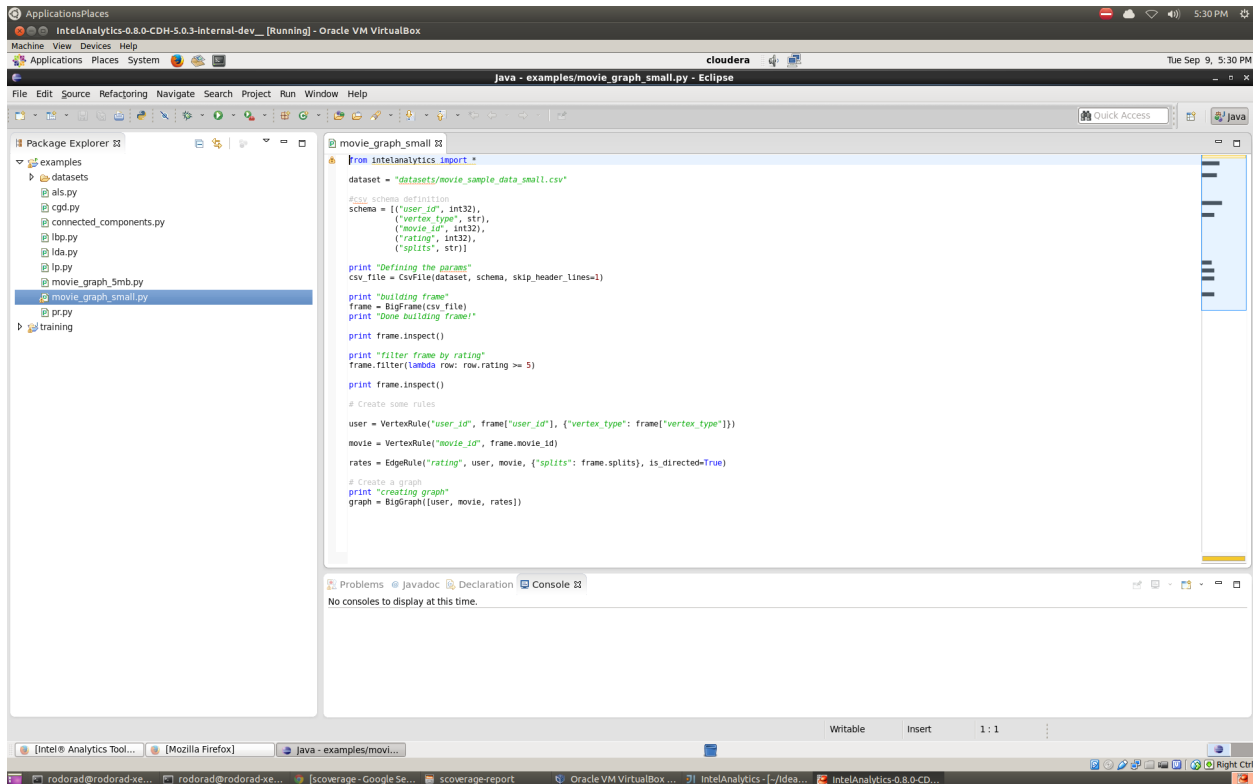


Fig. 18.15: Examining Example Scripts

18.10 Updating

Upon receipt of access and secret tokens, edit `/etc/yum.repos.d/ta.repo` and replace `myKey` and `mySecret`. Afterwards, it is recommended to run `yum` commands to check for and perform updates.

```
$ sudo [vi/vim] /etc/yum.repos.d/ta.repo

[Trusted Analytics repo]
name=Trusted Analytics yum repo
baseurl=https://s3-us-west-2.amazonaws.com/trustedanalytics-repo/
    release/latest/yum/dists/rhel/6
gpgcheck=0
priority=1
#enabled=0
s3_enabled=0
key_id=myKey
secret_key=mySecret
```

The `baseurl` line shown above has been broken for proper display in certain media. It should be entered as a single line with no spaces.

To check for new updates and see the difference between the new and installed version:

```
$ sudo yum info trustedanalytics-rest-server
```

To update:

```
$ sudo yum update trustedanalytics-rest-server
```

18.11 Common VM problems

- The VM doesn't have enough memory allocated.
- The TA REST server wasn't restarted after restart or boot.

BEST KNOWN METHODS (ADMIN)

Table of Contents

- *Configuration information*

19.1 Configuration information

19.1.1 Cluster Configuration

Table of Contents

- *Installation*
 - *Requirements*
- *Help*
 - *Common Options*
 - *Help — cluster-generate*
 - *Help — cluster-config*
 - *Help — cluster-explore*
 - *Help — cluster-push*
- *Generating CDH Configurations*
 - *Script: cluster-generate*
 - * *Unique Options*
 - * *Common Options*
 - * *Example*
 - * *cdh.json*
 - *Formulas*
 - * *Cluster*
 - *Accessing Individual Configurations*
 - * *Log*
 - * *Args*
 - * *CDH*
 - * *ATK*
 - *Tips*
- *Setting CDH Configurations*
 - *Common Options*
 - * *Example*
 - *user-cdh.json*
 - * *Finding Keys*
 - * *Converting Keys*
 - * *Verifying JSON*
 - * *Create cdh.json*
 - * *Push user-cdh.json*
 - * *merged-cdh.json*
- *Exploring CDH Configurations*
 - *Common Options*

The cluster configuration tool allows easy updating and optimization of any CDH cluster from the command line. With the use of optimization formulas you can have an optimized CDH cluster in a matter of seconds.

The tool does 3 things:

- Generate optimized configurations based on your cluster hardware.
- Easy updating, restarting, and configuration deployment of CDH services.
- Simple exploration of CDH cluster configurations.

Installation

Until the pip package is hosted, you will need to clone and install directly from source.

```
git clone git@github.intel.com:gao/cluster-config.git
cd cluster-config
[sudo] python setup.py install
```

Requirements

- Python 2.7
- `argparse`¹ `>= 1.3.0`
- `cm-api`² `== 10.0.0`
- Working CDH cluster for running the script

Help

After installing, you will have three executable scripts on your path: *Generating CDH Configurations* `cluster-config`, and `cluster-explore`.

Common Options

All three scripts have mostly overlapping configurations.

- **-host:** CDH ip or host name, always required
- **-port:** CDH port, default 7180, required if your CDH manager UI port is not 7180
- **-username:** CDH username, default 'admin', required if your CDH manager UI username is not 'admin'
- **-password:** CDH manager password, if not provided on the command line you will be asked at runtime
- **-cluster:** CDH cluster name and the name of the cluster if more than one cluster is managed by cloudera manager.
- **-path:** The path we will save and load files, like `cdh.json`, default is working directory
- **-log:** Log level, defaults to INFO

Running any of the scripts with the `-help` option will provide help text for all the command line options.

Help — cluster-generate

```
$ cluster-generate --help
usage: cluster-generate [-h] [--formula FORMULA] [--formula-args FORMULA_ARGS]
                        --host HOST [--port PORT] [--username USERNAME]
                        [--password PASSWORD] [--cluster CLUSTER]
                        [--path PATH] [--log {INFO,DEBUG,WARNING,FATAL,ERROR}]

Auto generate various CDH configurations based on system resources

optional arguments:
  -h, --help                show this help message and exit
  --formula FORMULA         Auto generation formula file. Defaults to
                             /home/rodorad/Documents/config/atk-
                             config/cluster_config/formula.py
  --formula-args FORMULA_ARGS
                             Auto generation formula arguments to possibly override
                             global configurations.
  --host HOST               Cloudera Manager Host
```

¹<https://docs.python.org/2.7/library/argparse.html>

²https://github.com/cloudera/cm_api

```
--port PORT           Cloudera Manager Port
--username USERNAME   Cloudera Manager User Name
--password PASSWORD   Cloudera Manager Password
--cluster CLUSTER     Cloudera Manager Cluster Name if more than one cluster
                      is managed by Cloudera Manager.
--path PATH           Directory where we can save/load configurations files.
                      Defaults to working directory
                      /home/rodorad/Documents/config/atk-config
--log {INFO,DEBUG,WARNING,FATAL,ERROR}
                      Log level [INFO|DEBUG|WARNING|FATAL|ERROR]
```

Help — cluster-config

```
$ cluster-config --help
usage: cluster-config [-h] --update-cdh {no,yes} --restart-cdh {no,yes}
                      [--conflict-merge {interactive,user,generated}] --host
                      HOST [--port PORT] [--username USERNAME]
                      [--password PASSWORD] [--cluster CLUSTER] [--path PATH]
                      [--log {INFO,DEBUG,WARNING,FATAL,ERROR}]

Process cl arguments to avoid prompts in automation

optional arguments:
  -h, --help                show this help message and exit
  --update-cdh {no,yes}     Should we update CDH with all configurations in
                           cdh.json/user-cdh.json?
  --restart-cdh {no,yes}    Should we restart CDH services after configuration
                           changes
  --conflict-merge {interactive,user,generated}
                           When encountering merge conflicts between the
                           generated configuration() and the user configuration()
                           what value should we default to? The 'user',
                           'generated', or 'interactive' resolution
  --host HOST               Cloudera Manager Host
  --port PORT               Cloudera Manager Port
  --username USERNAME       Cloudera Manager User Name
  --password PASSWORD       Cloudera Manager Password
  --cluster CLUSTER         Cloudera Manager Cluster Name if more than one cluster
                           is managed by Cloudera Manager.
  --path PATH               Directory where we can save/load configurations files.
                           Defaults to working directory /home/SOME_USER
  --log {INFO,DEBUG,WARNING,FATAL,ERROR}
                           Log level [INFO|DEBUG|WARNING|FATAL|ERROR]
```

Help — cluster-explore

```
$ cluster-explore --help
usage: cluster-explore [-h] [--dump DUMP] --host HOST [--port PORT]
                      [--username USERNAME] [--password PASSWORD]
                      [--cluster CLUSTER] [--path PATH]
                      [--log {INFO,DEBUG,WARNING,FATAL,ERROR}]
```

Process cl arguments to avoid prompts in automation

optional arguments:

```
-h, --help            show this help message and exit
--dump DUMP           If you want to dump all configs without asking pass
                      'yes'. Defaults to 'no'.
--host HOST           Cloudera Manager Host
--port PORT           Cloudera Manager Port
--username USERNAME   Cloudera Manager User Name
--password PASSWORD   Cloudera Manager Password
--cluster CLUSTER     Cloudera Manager Cluster Name if more than one cluster
                      is managed by Cloudera Manager.
--path PATH           Directory where we can save/load configurations files.
                      Defaults to working directory
                      /home/SOME_USER/
--log {INFO,DEBUG,WARNING,FATAL,ERROR}
                      Log level [INFO|DEBUG|WARNING|FATAL|ERROR]
```

Help — cluster-push

Combines cluster-generate and cluster-config

```
$ cluster-push --help
usage: cluster-push [-h] [--formula FORMULA] [--formula-args FORMULA_ARGS]
                  --update-cdh {no,yes} --restart-cdh {no,yes}
                  [--conflict-merge {interactive,user,generated}] --host
HOST [--port PORT] [--username USERNAME]
                  [--password PASSWORD] [--cluster CLUSTER] [--path PATH]
                  [--log {INFO,DEBUG,WARNING,FATAL,ERROR}]
```

Auto generate various CDH configurations based on system resources

optional arguments:

```
-h, --help            show this help message and exit
--formula FORMULA     Auto generation formula file. Defaults to
                      /home/rodorad/Documents/config/atk-
                      config/cluster_config/formula.py
--formula-args FORMULA_ARGS
                      Auto generation formula arguments to possibly override
                      global configurations.
--update-cdh {no,yes} Should we update CDH with all configurations in
                      cdh.json/user-cdh.json?
--restart-cdh {no,yes} Should we restart CDH services after configuration
                      changes
--conflict-merge {interactive,user,generated}
                      When encountering merge conflicts between the
                      generated configuration() and the user configuration()
                      what value should we default to? The 'user',
                      'generated', or 'interactive' resolution
--host HOST           Cloudera Manager Host
--port PORT           Cloudera Manager Port
--username USERNAME   Cloudera Manager User Name
--password PASSWORD   Cloudera Manager Password
--cluster CLUSTER     Cloudera Manager Cluster Name if more than one cluster
                      is managed by Cloudera Manager.
```

```
--path PATH          Directory where we can save/load configurations files.
                      Defaults to working directory
                      /home/rodorad/Documents/config/atk-config
--log {INFO,DEBUG,WARNING,FATAL,ERROR}
                      Log level [INFO|DEBUG|WARNING|FATAL|ERROR]
```

Generating CDH Configurations

Script: cluster-generate

Description: used for generating optimized CDH configurations

Unique Options

- **-formula:** path to formula file if not using the default packaged formula.
- **-formula-args:** path to formula arguments to possibly override constants.

Common Options If you forget the command line options run the script with **-help** or visit the [help section](#).

Providing only CDH connection details should be sufficient for generating optimized configurations since **-formula** is provided with a default.

The default formula can be found in the repository at *cluster_cofig/formula.py*.

Example

```
$ cluster-generate --host CLOUDERA_HOST --port CLOUDERA_PORT --username CLOUDERA_USER
What is the Cloudera manager password?
--INFO cluster selected: SOME_CLUSTER
--INFO using formula: cluster_config/formula.py
--INFO Wrote CDH configuration file to: /home/some-user/cdh.json
--INFO Wrote ATK generated configuration file to: /home/some-user/generated.conf
```

After providing the Cloudera manager password, all the cluster details will be extracted and provided to the formula to calculate an optimized configuration which gets saved to *cdh.json*.

The generated.conf file will contain ATK specific configurations. It can be ignored, if you don't intend to use the ATK server for analytics.

The optimized configurations are saved to a file to allow users to view and verify the configurations before they get saved to CDH with the *cluster-config* script.

Although we take great care to make sure the default formula will work for a majority of use cases, you should verify the settings to make sure your cluster will not be adversely affected.

Next we will read *cdh.json* and update CDH.

cdh.json Here is a chd.json file generated on a 4 node cluster

- 1 node having Cloudera manager
- 3 nodes running as workers nodes
- 32 cores
- 252 gigabytes of memory

```
{
  "HBASE": {
    "REGIONSERVER": {
      "REGIONSERVER_BASE": {
        "HBASE_REGIONSERVER_JAVA_HEAPSIZE": 26430567975
      }
    }
  },
  "YARN": {
    "GATEWAY": {
      "GATEWAY_BASE": {
        "MAPREDUCE_MAP_JAVA_OPTS_MAX_HEAP": 4129776246,
        "MAPREDUCE_MAP_MEMORY_MB": 5368709120,
        "MAPREDUCE_REDUCE_JAVA_OPTS_MAX_HEAP": 8259552492,
        "MAPREDUCE_REDUCE_MEMORY_MB": 10737418240,
        "YARN_APP_MAPREDUCE_AM_MAX_HEAP": 4129776246,
        "YARN_APP_MAPREDUCE_AM_RESOURCE_CPU_VCORES": 1,
        "YARN_APP_MAPREDUCE_AM_RESOURCE_MB": 5368709120
      }
    },
    "NODEMANAGER": {
      "NODEMANAGER_BASE": {
        "YARN_NODEMANAGER_RESOURCE_CPU_VCORES": 32,
        "YARN_NODEMANAGER_RESOURCE_MEMORY_MB": 180388626432
      }
    },
    "RESOURCEMANAGER": {
      "RESOURCEMANAGER_BASE": {
        "YARN_SCHEDULER_INCREMENT_ALLOCATION_MB": 536870912,
        "YARN_SCHEDULER_MAXIMUM_ALLOCATION_MB": 180388626432,
        "YARN_SCHEDULER_MAXIMUM_ALLOCATION_VCORES": 32,
        "YARN_SCHEDULER_MINIMUM_ALLOCATION_MB": 5368709120
      }
    }
  }
}
```

Formulas

It is possible to create your optimization formulas or copy the default formula and make your own tweaks. The formula files are regular python scripts. While the default formula is called ‘formula.py’ there are no restrictions.

The formula files are executed with `python’s exec file function`³.

Cluster The cluster variable will contain all installed service, role, config group and configuration details from the CDH cluster we are currently connected to.

Lets look at a code sample from formula.py to see how the cluster variable is being used.

```
...
hosts = cluster.yarn.nodemanager.hosts.all()
max_cores = 0
max_memory = 0
for key in hosts:
    if hosts[key].numCores > max_cores:
```

³<https://docs.python.org/2/library/functions.html#execfile>

```

    max_cores = hosts[key].numCores
    if hosts[key].totalPhysMemBytes > max_memory:
        max_memory = hosts[key].totalPhysMemBytes
...

```

On the first line with `cluster.yarn.nodemanager.hosts.all()` we are retrieving all the details for every host running the yarn node manager role. The returned object will be a list of cloudera [apihosts](#)⁴.

With the same notation you can retrieve all the details for hosts running the hbase region server role or all zookeeper servers.

- `cluster.hbase.regionserver.hosts.all()`
- `cluster.zookeeper.server.hosts.all()`

You can access all CDH services, roles, configs, and hosts with the same notation.

The pattern you want to follow is

- `cluster.SERVICE.ROLE.CONFIG_GROUP.CONFIG`
- `cluster.SERVICE.ROLE.hosts.all`
- **SERVICE** is any valid CDH [service](#)⁵
- **ROLE** is any valid CDH [role](#)⁶
- **CONFIG_GROUP** is any valid config group from [cluster-explore](#) script. A good example of a configuration group would be 'gateway_base' because just about every service has this role.
- **CONFIG** is any valid config from [cluster-explore](#) script. When accessing CDH services all attributes are lowercase while the keys displayed by the [cluster-explore](#) script are uppercase.

Accessing Individual Configurations Accessing individual configurations is easy but you must know the configuration name. You will need to run the [cluster-explore](#) script to find the full configuration name.

Here is a config key that was found with [cluster-explore](#).

```

- name: HBASE_REGIONSERVER_JAVA_HEAPSIZE
- description: Maximum size in bytes for the Java Process heap memory. Passed to Java -Xmx.
- key: HBASE.REGIONSERVER.REGIONSERVER_BASE.HBASE_REGIONSERVER_JAVA_HEAPSIZE
- value: 26430567975

```

To retrieve the following value with the cluster variable make the key all lower case and prefix cluster.

cluster.hbase.regionserver.regionserver_base.hbase_regionserver_java_heapsize.value

If you look at the implementation for the 'cluster' objects class you will notice that you can set all CDH configurations directly i would cuation against it. Saving the computed configurations to the '**cdh**' dictionary will allow the user to review the configurations when they get saved to *cdh.json* and enable the saving of snapshot for administrative purposes.

Log The log variable has four methods available for sending log messages to the command line:

- `log.info`
- `log.error`

⁴http://cloudera.github.io/cm_api/apidocs/v10/ns0_apiHost.html

⁵http://cloudera.github.io/cm_api/apidocs/v10/path__clusters_-clusterName-_services.html

⁶http://cloudera.github.io/cm_api/apidocs/v10/path__clusters_-clusterName-_services_-serviceName-_roles.html

- `log.warning`
- `log.fatal`: does a `sys.exit(1)` after logging its message. All the methods take a single string as an argument.

Args `Args` is a dictionary of values provided by the user. The `args` dictionary gives us the ability to consider the users Constants when calculating configurations.

When calculating optimized configurations you often need constants to set thresholds. The default `formula.py` has several constants to allow for fine tuning.

..code:

```
NUM_THREADS = 1 # This should be set to the maximum number of multi-tenant users
OVER_COMMIT_FACTOR = 1.30
MAX_JVM_MEMORY = 32768 * MiB
MEM_FRACTION_FOR_HBASE = 0.20
MEM_FRACTION_FOR_OTHER_SERVICES = 0.20
MAPREDUCE_JOB_COUNTERS_MAX = 500
```

If the value for `'NUM_THREADS'` is given on the command line, that value is received in `'args["NUM_THREADS"]'` allowing for the decision of whether to use the default or the specified value.

CDH The `'cdh'` dictionary is the variable where the calculated CDH configurations are saved with the dictionary key being the full key to the path found with the `cluster-explore` script.

When the script has finished running, the contents of the `'cdh'` dictionary are saved to `cdh.json` so the configurations can be reviewed before updating CDH.

For example, if you wanted to save your optimized value for the hbase heap you would do this:

```
cdh[ 'HBASE.REGIONSERVER.REGIONSERVER_BASE.HBASE_REGIONSERVER_JAVA_HEAPSIZE ' ] = SOME_MEMORY_SETTING
```

Once the `execfile` has finished running, the `'cdh'` dictionary is exported to `cdh.json` which is read by the `cluster-config` or `help_cluster-push` scripts to update CDH.

ATK Much like the `'cdh'` dictionary, the `'atk'` dictionary is the place where the calculated ATK configurations are saved. If the ATK, is not being used, the dictionary can be safely ignored.

The key for every value in the `'atk'` dictionary would need to be a valid `atk configuration key`⁷.

Tips

When saving optimized configurations be aware of the expected format. For example, a memory setting for `yarn`, `hbase`, and `spark` may require different formats. `Yarn` may expect configuration values expressed in mega bytes, while `hbase` and `spark` expect values expressed in bytes.

Unfortunately, the CDH REST API does not always throw formatting errors. If there are formatting issues, login to CDH console and look for configuration warnings concerning the affected services.

Setting CDH Configurations

Script: `cluster-config`

Description: Used for updating CDH with the optimized configurations

⁷<https://github.com/intel-data/atk/blob/master/conf/examples/application.conf.tpl>

Unique Options:

- **-update-cdh**: Whether to update CDH configurations. Either yes or no.
- **-restart-cdh**: Whether to restart CDH after updating its configuration. Either yes or no.
- **-conflict-merge**: Conflict resolution preference when encountering key conflicts between `cdh.json` and `user-cdh.json`. Defaults to `user-cdh.json`. Valid values [user, generated, interactive]

Common Options

If you forget the command line options run the script with **-help** or visit the [help section](#).

Warning: The **update-cdh**, and **restart-cdh** options are provided as fail-safe to keep unwanted changes from potentially breaking a working cluster, and to keep restarts from stopping long running jobs.

Example

```
$ cluster-config --host CLOUDERA_HOST --port CLOUDERA_PORT --username CLOUDERA_USER --update-cdh yes -
What is the Cloudera manager password?
--INFO cluster selected: SOME_CLUSTER
--INFO Reading CDH config file: /home/some-user/cdh.json
--WARNING Couldn't open json file: /home/rodorad/user-cdh.json
--INFO Writting merged CDH config file: /home/some-user/merged-cdh.json
--INFO Updating config group: REGIONSERVER_BASE
--INFO Updated 1 configuration/s.
Restarting service : "HBASE"
. . . . .

Deploying configuration for all HBASE roles
.

--INFO Updating config group: NODEMANAGER_BASE
--INFO Updated 2 configuration/s.
--INFO Updating config group: GATEWAY_BASE
--INFO Updated 7 configuration/s.
--INFO Updating config group: RESOURCEMANAGER_BASE
--INFO Updated 4 configuration/s.
Restarting service : "YARN"
. . . . .

Deploying configuration for all YARN roles
.
```

After connecting to Cloudera Manager, `cdh.json` (generated in the [previous step](#) and `user-cdh.json` (if available) are read.

The file `user-cdh.json` has any user overrides and/or any additional CDH configurations needing to be set. For this example, the `user-cdh.json` file was omitted.

When **-update-cdh**, and **-restart-cdh** are set to 'yes', the CDH needs to update, and restart CDH with the configurations from `cdh.json/user-cdh.json`.

user-cdh.json

The *user-cdh.json* file is where to set any CDH configurations. As long the file is in the working directory or in directory specified by `–path`, it will be read and pushed to CDH by the *cluster-config* script.

A sample *user-cdh.json* file looks like a *cdh.json* file. It can have the same or different keys as a *cdh.json* file.

Finding Keys For a simple demonstration, look for any key with the *cluster-explore* script to add to our *user-cdh.json* file. This example shows **YARN.JOBHISTORY.JOBHISTORY_BASE.YARN_JOBHISTORY_BIND_WILDCARD**.

```
config:
- Name: YARN_JOBHISTORY_BIND_WILDCARD
- Description: If enabled, the JobHistory Server binds to the wildcard address ("0.0.0.0") on all of
- Key: YARN.JOBHISTORY.JOBHISTORY_BASE.YARN_JOBHISTORY_BIND_WILDCARD
- Value: false
```

This is what it looks like in CDH:

Bind JobHistory Server to Wildcard Address JobHistory Server Default Group ☐

Converting Keys To save the key to *user-cdh.json* format, the key needs to be split at the each period with each element being nested inside the previous element.

key: **YARN.JOBHISTORY.JOBHISTORY_BASE.YARN_JOBHISTORY_BIND_WILDCARD** would convert to:

```
{
  "YARN": {
    "JOBHISTORY": {
      "JOBHISTORY_BASE": {
        "YARN_JOBHISTORY_BIND_WILDCARD": "true"
      }
    }
  }
}
```

Lets go ahead and add another key that will conflict with our *cdh.json* file to see how conflict resolution works.

I added **'YARN_NODEMANAGER_RESOURCE_CPU_VCORES'** which is set to 32 cores in my *cdh.json*. Add the key and subtract a couple of cores from the value:

```
{
  "YARN": {
    "JOBHISTORY": {
      "JOBHISTORY_BASE": {
        "YARN_JOBHISTORY_BIND_WILDCARD": "true"
      }
    },
    "NODEMANAGER": {
      "NODEMANAGER_BASE": {
        "YARN_NODEMANAGER_RESOURCE_CPU_VCORES": 30
      }
    }
  }
}
```

```
}
}
```

Once you have your *user-cdh.json* file make sure you save it to your working directory.

Verifying JSON All *user-cdh.json* files need to contain only valid json text. If you get parsing errors or have any doubts about your formatting, you can use a [json lint tool](http://jsonlint.com/)⁸ to format and error check your json text.

Create cdh.json Once you've saved *user-cdh.json*, run *Generating CDH Configurations* to create a *cdh.json* file, or verify you have *_cdh.json_* file in the same working directory where you saved *_user-cdh.json_* file.

Push user-cdh.json Running the *cluster-config* script will automatically push *user-cdh.json* configurations to CDH after merging with *cdh.json*.

You should see log messages notifying you that *user-cdh.json* was loaded.

```
$ cluster-config --host SOME_USER --update-cdh yes --restart-cdh no
What is the Cloudera manager password?
--INFO cluster selected: SOME_CLUSTER
--INFO Reading CDH config file: /home/SOME_USER/cdh.json
--INFO Reading user CDH config file: /home/SOME_USER/user-cdh.json
--INFO conflict resolution: first

Key merge conflict: YARN.NODEMANAGER.NODEMANAGER_BASE.YARN_NODEMANAGER_RESOURCE_CPU_VCORES
[user]User value: 30
[auto]Auto generated value: 32
Optionally you can accept [au] all user values or [ag] all auto generated values.
--INFO Auto resolving conflicts. Defaulting to user value
--INFO Writting merged CDH config file: /home/rodorad/merged-cdh.json
--INFO Updating config group: REGIONSERVER_BASE
--INFO Updated 1 configuration/s.
--INFO Updating config group: NODEMANAGER_BASE
--INFO Updated 2 configuration/s.
--INFO Updating config group: JOBHISTORY_BASE
--INFO Updated 1 configuration/s.
--INFO Updating config group: RESOURCEMANAGER_BASE
--INFO Updated 4 configuration/s.
--INFO Updating config group: GATEWAY_BASE
--INFO Updated 7 configuration/s.
```

In above example we see notifications when we load *user-cdh.json* and how conflicts were resolved.

The default behavior when encountering conflicting keys is to resolve them to the user value. You can change the default behavior on the command line with the **–conflict-merge** option.

We also see that a *merged-cdh.json* file was created. This file gets created when the *user-cdh.json* and *cdh.json* files are merged. It is a record of all the configurations that got pushed to CDH after resolving key conflicts.

After pushing our changes our CDH value is updated.

Bind JobHistory Server to Wildcard Address

JobHistory Server Default Group ☒ 

⁸<http://jsonlint.com/>

merged-cdh.json sample merged-cdh.json file

```
{
  "HBASE": {
    "REGIONSERVER": {
      "REGIONSERVER_BASE": {
        "HBASE_REGIONSERVER_JAVA_HEAPSIZE": 26430567975
      }
    }
  },
  "YARN": {
    "GATEWAY": {
      "GATEWAY_BASE": {
        "MAPREDUCE_MAP_JAVA_OPTS_MAX_HEAP": 4129776246,
        "MAPREDUCE_MAP_MEMORY_MB": 5368709120,
        "MAPREDUCE_REDUCE_JAVA_OPTS_MAX_HEAP": 8259552492,
        "MAPREDUCE_REDUCE_MEMORY_MB": 10737418240,
        "YARN_APP_MAPREDUCE_AM_MAX_HEAP": 4129776246,
        "YARN_APP_MAPREDUCE_AM_RESOURCE_CPU_VCORES": 1,
        "YARN_APP_MAPREDUCE_AM_RESOURCE_MB": 5368709120
      }
    },
    "JOBHISTORY": {
      "JOBHISTORY_BASE": {
        "YARN_JOBHISTORY_BIND_WILDCARD": "True"
      }
    },
    "NODEMANAGER": {
      "NODEMANAGER_BASE": {
        "YARN_NODEMANAGER_RESOURCE_CPU_VCORES": 30,
        "YARN_NODEMANAGER_RESOURCE_MEMORY_MB": 180388626432
      }
    },
    "RESOURCEMANAGER": {
      "RESOURCEMANAGER_BASE": {
        "YARN_SCHEDULER_INCREMENT_ALLOCATION_MB": 536870912,
        "YARN_SCHEDULER_MAXIMUM_ALLOCATION_MB": 180388626432,
        "YARN_SCHEDULER_MAXIMUM_ALLOCATION_VCORES": 32,
        "YARN_SCHEDULER_MINIMUM_ALLOCATION_MB": 5368709120
      }
    }
  }
}
```

Exploring CDH Configurations

Script: **cluster-explore**

Description: Browse CDH configurations by service, role, and configuration group or dumped all at once to the screen. Helps find the configuration keys if you want to write your own *user-cdh.json* file or if you want to write your own formula file.

Unique Options:

- **-dump**: if 'yes' dumps all the configurations for every service to stdout.

Common Options

If you forget the command line options run the script with **-help** or visit the [help section](#).

Finding configurations is quite difficult without an appropriate tool. The cluster explore script solves this very problem.

It has two run modes:

1. Print all the configurations for every service to the screen.
2. Interactively print all the configurations for a specific service, role, and configuration group.

To invoke option 1 you can answer 'yes' to the **-dump** command line option or answer yes when prompted on the command line.

```
$ cluster-explore --host SOME_HOST --port SOME_PORT --username SOME_USERNAME
What is the Cloudera manager password?
--INFO cluster selected: SOME_CLUSTER
dump all configs[yes or no]: yes

...
- config name: ENABLE_CONFIG_ALERTS
- config description: When set, Cloudera Manager will send alerts when this entity's configuration ch
- config key: SPARK_ON_YARN.SPARK_YARN_HISTORY_SERVER.SERVER_BASE.ENABLE_CONFIG_ALERTS
- config value: false

- config name: PROCESS_AUTO_RESTART
- config description: When set, this role's process is automatically (and transparently) restarted in
- config key: SPARK_ON_YARN.SPARK_YARN_HISTORY_SERVER.SERVER_BASE.PROCESS_AUTO_RESTART
- config value: false
...
```

To invoke option 2, run the script interactively. Answer “no” or hit enter when prompted to dump the configurations.

```
$ cluster-explore --host SOME_HOST --port SOME_PORT --username SOME_USERNAME
What is the Cloudera manager password?
--INFO cluster selected: SOME_CLUSTER
dump all configs[yes or no]: no

Available service types on cluster: 'cluster'
Pick a service
Id 1 service: HDFS
Id 2 service: HBASE
Id 3 service: ZOOKEEPER
Id 4 service: YARN
Id 5 service: SPARK_ON_YARN
Enter service Id : 4
Selected YARN

Available role types on service: 'YARN'
Pick a role
Id 1 role: NODEMANAGER
Id 2 role: JOBHISTORY
Id 3 role: RESOURCEMANAGER
Id 4 role: GATEWAY
Enter role Id : 4
Selected GATEWAY

Available config group types on role: 'GATEWAY'
```

```

Pick a config group
Id 1 config group: GATEWAY_BASE
Enter config group Id : 1
Selected GATEWAY_BASE

...

config:
- name: MAPRED_OUTPUT_COMPRESSION_CODEC
- description: For MapReduce job outputs that are compressed, specify the compression codec to use. W
- key: YARN.GATEWAY.GATEWAY_BASE.MAPRED_OUTPUT_COMPRESSION_CODEC
- value: org.apache.hadoop.io.compress.DefaultCodec

config:
- name: MAPRED_REDUCE_PARALLEL_COPIES
- description: The default number of parallel transfers run by reduce during the copy (shuffle) phase
- key: YARN.GATEWAY.GATEWAY_BASE.MAPRED_REDUCE_PARALLEL_COPIES
- value: 10

config:
- name: MAPREDUCE_JOB_UBERTASK_MAXMAPS
- description: Threshold for number of maps, beyond which a job is considered too big for ubertask op
- key: YARN.GATEWAY.GATEWAY_BASE.MAPREDUCE_JOB_UBERTASK_MAXMAPS
- value: 9

...

```

For every configuration key you will see

- configuration name: Normalized configuration name to account for some weirdness with CDH configuration names like slashes and dashes.
- configuration description: This will be the description from CDH UI.
- full configuration key path: The format for a key is `SERVICE_TYPE.ROLE_TYPE.CONFIG_GROUP_NAME.CONFIGURATION_KEY`. While composed mostly of types the `CONFIG_GROUP_NAME` is normalized to remove service names which can be anything.
- value: The value of the configuration.

19.1.2 Giraph Performance Tuning and Benchmarking

Apache Giraph is a Hadoop-based graph-processing framework that is modeled after Google's Pregel. Giraph jobs are memory-hungry applications because it attempts to load the entire graph in-memory. Giraph jobs have no reducers. To improve memory-related issues:

- Allocate all available memory to the job, in other words, set the total JVM heap size available to mappers in `$HADOOP_HOME/conf/mapred-site` (`mapreduce.tasktracker.map.tasks.maximum` * maximum heap size in `mapred.map.child.java.opts`) so it does not exceed available memory for the node, and leaves room for other daemons running on the nodes, for example, datanode, regionserver, and tasktracker.
- Allocate at least 4GB to each map tasks, but this value depends heavily on the size of the graph.
- The maximum number of workers available is the number of map slots available in the cluster - 1 (in other words, the number of nodes times the `mapreduce.tasktracker.map.tasks.maximum` - 1). One master and the rest are slaves.
- When writing Giraph jobs, keep the compute function small and minimize vertex memory footprint. There is a patch for spilling data to disk but it does not appear to be working very well at the moment.

To solve performance problems, try upgrading to the latest version of Giraph, which might include patches that address performance concerns. For example, Facebook submitted a large number of patches that allow Giraph to run at scale [http link⁹](#). Some memory optimizations to try are:

- Enable ByteArrayPartition store that Facebook implemented to optimize memory usage in Giraph using the option **-Dgiraph.partitionClass=org.apache.giraph.partition.ByteArrayPartition**. This option significantly decreases the amount of memory needed to run Giraph jobs. Giraph's default partition store is the SimplePartitionStore (which stores the graph as Java objects). The ByteArrayPartition store keeps only a representative vertex that serializes/deserializes on the fly thereby saving memory. Be careful to use this option, because some algorithms may fail with this option on.
- If messages are stored as complex objects, try enabling the option **-Dgiraph.useMessageSizeEncoding=true**.
Note: This option will cause jobs to crash if the messages are simple objects.

Other Giraph options are available at <http://giraph.apache.org/options.html>. These options can be set using **-D<option=value>** in the command for running the Hadoop job. For large clusters, “The Hadoop Real World Solutions Cookbook, Owens et al. 2013” recommends controlling the number of messaging threads used by each worker to communicate with other workers, and increasing the maximum number of messages flushed in bulk.

19.1.3 HBase Performance Tuning

Two comprehensive sources for performance tuning in HBase are:

- [HBase reference guide¹⁰](#)
- [The HBase Administration Cookbook¹¹](#)

Some of the recommendations for production clusters are detailed below. Remember to sync these changes to all servers in the cluster.

1. Increase ulimit settings. The default Linux setting for maximum number of file descriptors for a user is 1024 which is too low. The number of file descriptors (nofile) and open processes for the user running the Hadoop/HBase daemons should be set to a minimum of 32768, otherwise the system will experience problems under heavy load. Instructions for changing */etc/security/limits.conf* are available here: <http://hbase.apache.org/book.html#ulimit>
2. Garbage collection. Pauses due to garbage collection can impact performance. For latency-sensitive applications, Cloudera recommends **-XX:+UseParNewGC** and **-XX:+UseConcMarkSweepGC** in *hbase-env.sh*. The MemStore-Local Allocation buffers (MSLAB) which reduce heap fragmentation under heavy write-load are enabled by default in HBase. The MSLAB setting is **hbase.hregion.memstore.mslab.enabled** in *hbase-site.xml*. <http://blog.cloudera.com/blog/2011/02/avoiding-full-gcs-in-hbase-with-memstore-local-allocation-buffers-part-1>
3. Enable column family compression. Compression boosts performance by reducing the size of store files thereby reducing I/O <http://hbase.apache.org/book.html#compression>. The Snappy library from Google is faster than LZO and easier to install. Instructions for installing Snappy are available at: <http://hbase.apache.org/book.html#d0e24641> Once installed, turn on column family compression when creating the table, for example:

```
create 'mytable', {NAME => 'mycolumnfamily1', COMPRESSION => 'SNAPPY'}
```

4. Avoid update blocking for write-heavy workloads. If the cluster has enough memory, increase **hbase.hregion.memstore.block.multiplier** and **hbase.hstore.blockingStoreFiles** in *hbase-site.xml* to avoid

⁹<http://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920>

¹⁰http://hbase.apache.org/book.html#important_configurations

¹¹<http://library.intel.com/Catalog/CatalogItemDetails.aspx?id=225983>

blocking in write-heavy workloads. This setting needs to be tuned carefully because it increases the number of files to compact. See [The HBase Administration Cookbook](#)¹² (Chapter 9)

5. Increase region server handler count. If the workload has many concurrent clients, increasing the number of handlers can improve performance. However, too many handlers will lead to out of memory errors. See [The HBase Administration Cookbook](#)¹³ (Chapter 9)

Other recommendations include turning off automatic compactions and doing it manually:
<http://hbase.apache.org/book.html#managed.compactions>.

¹²<http://library.intel.com/Catalog/CatalogItemDetails.aspx?id=225983>

¹³<http://library.intel.com/Catalog/CatalogItemDetails.aspx?id=225983>

WINDOWS PYTHON SETUP

1. Download Anaconda with Python 2.7.
2. Install
 - By default, Anaconda installs to the user's AppData hidden folder. It's better to put this in a more accessible location, like `c:\anaconda`. This is the only change from the default installation necessary.
3. Open a command prompt.
4. Run the command: `conda create -n trustedanalytics-python python=2.7 numpy=1.8 requests=2.3 ordereddict=1.1`. This creates a virtual Python environment that mimics the cluster's configuration.

INTEGRATED DEVELOPMENT ENVIRONMENTS

21.1 Eclipse Setup Instructions

Table of Contents

- *Prerequisites*
- *Setup*

21.1.1 Prerequisites

These instructions assume you have already installed:

- *Python 2.7*.
- Trusted Analytics *Python REST Client* and required dependencies.
- Eclipse Standard.

Trusted Analytics should work with any version of Eclipse, but these instructions were tested with Eclipse Standard Version 4.4 Luna.

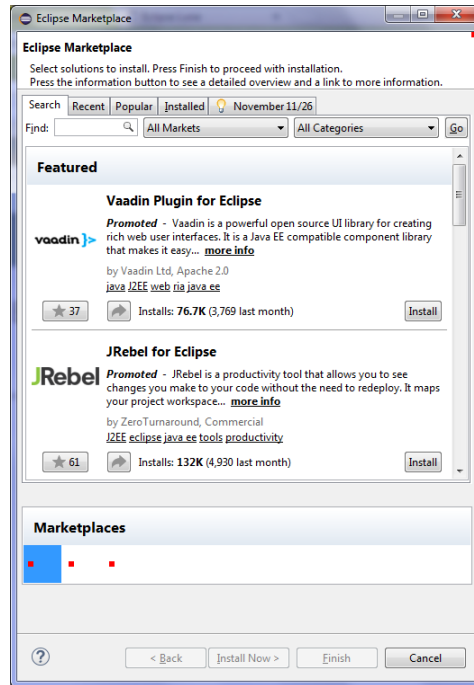
If you are on a corporate network, you may need to configure proxy settings in Eclipse before beginning (see [Eclipse Help](#)¹).

21.1.2 Setup

1. Download, install, and start the [Eclipse IDE](#)².
2. Choose *Help* → *Eclipse Marketplace*.

¹http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.jpt.doc.user%2Ftips_and_tricks.htm

²<http://www.eclipse.org/>



If the marketplace screen does not come up, you may need to configure proxy settings (see [Eclipse Help³](#)).

3. Search for “PyDev” and choose *PyDev - Python IDE for Eclipse 3.6.0* or newer version.
 - (a) Choose *Confirm* button.
 - (b) Choose *Accept* when prompted for license agreement.
 - (c) If prompted “Do you trust these certificates?”. Select “Brainwy Software; PyDev; Brainwy” and choose the *OK* button.
 - (d) When prompted to restart Eclipse, choose the *Yes* button.
4. Choose the default Workspace.
5. Choose *File* → *New* → *Project...*
 - (a) Choose the *PyDev* folder and *PyDev Project* and choose the *Next* button.
 - (b) Give your project a name, for example “myproject”.
 - (c) Choose version 2.7.
 - (d) Choose *Please configure an interpreter before proceeding*.
 - i. Choose *Manual Configure*.
 - ii. Choose the *New* button.
 - iii. Browse for Python 2.7. On RedHat and Centos this is probably `/usr/bin/python`.
 - iv. Choose the *Ok* button.
 - v. Choose the *Ok* button.
 - (e) Select the interpreter you just setup from the Interpreter drop-down.
 - (f) Choose the *Finish* button.

³http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Ftips_and_tricks.htm

- (g) When prompted “This kind of project is associated with the PyDev perspective. Do you want to open this perspective now?” choose *Yes*.
- 6. Right click your project folder, for example, “myproject”.
 - (a) Choose *Properties*.
 - (b) Choose *PyDev - PYTHONPATH* in the left hand pane.
 - (c) Choose the *External Libraries* tab.
 - (d) Choose *Add source folder* button.
 - (e) Browse for the Trusted Analytics Python REST Client libraries. On RedHat and Centos these are found under ‘/usr/lib/trustedanalytics/rest-client/python’.
 - (f) Choose the *OK* button.
- 7. Right click your project folder, for example, “myproject”.
 - (a) Choose *New → Source Folder*.
 - (b) Give it the name “src” and choose the *Finish* button.
- 8. Right click *src* folder and choose *New → File*.
 - (a) Give the file name ‘test.py’.
 - (b) If prompted, confirm the default settings for PyDev by choosing *OK*.
 - (c) Close the *Help keeping PyDev alive* dialog, if it appears.
 - (d) Type the following code into test.py:

```
import trustedanalytics as ta
ta.server.host = "correct host name or IP address"
ta.connect()
ta.server.ping()
```

- 9. Choose *Run → Run*.
 - (a) Choose *Python Run* and choose the *OK* button, you should see the output:

```
Successful ping to Trusted Analytics ATK at http://localhost:9099/info
```

- 10. Next take a look at the included examples.

21.2 IntelliJ IDEA Setup Instructions

Table of Contents

- [Prerequisites](#)
- [Setup](#)

21.2.1 Prerequisites

These instructions assume you have already installed:

- Python 2.7:

- Trusted Analytics Python REST Client and required dependencies.
- IntelliJ IDEA⁴.

Verify the Python plugin is enabled in IntelliJ by choosing *File* → *Settings*, searching for *Python*, and choosing *Plugins* from the pane on the left-hand side.

Trusted Analytics should work with any version of IntelliJ IDEA but these instructions were tested with IntelliJ IDEA 13.1.3 Ultimate.

21.2.2 Setup

1. Select *New Project* on IntelliJ IDEA's initial screen.
2. Select *Python* as the project type and choose *Next*.
 - (a) Choose *Next* leaving *Create project form template* unchecked.
 - (b) Choose *Python 2.7* as the Python Interpreter and choose the *Next* button.
 - i. If *Python 2.7* does not appear in the list you will need to configure a Python 2.7 Interpreter.
 - A. Choose the *Configure* button.
 - B. Choose the plus sign +.
 - C. Choose *Python SDK*.
 - D. Choose *Add Local*.
 - E. Browse for your local Python 2.7 installation. On RedHat and Centos this is probably `/usr/bin/python`.
 - F. Choose the *OK* button.
 - (c) Give your project a name, e.g *myproject*.
 - (d) Choose the *Finish* button.
3. Choose *File* → *Project Structure*.
 - (a) Make sure *Python 2.7* is selected as the Project SDK and choose *Apply*.
 - (b) Choose *Libraries* in the left hand pane.
 - (c) Choose the plus sign +.
 - (d) Choose *Java* and browse to the Trusted Analytics Python REST Client libraries. On RedHat and Centos these are found under `'/usr/lib/trustedanalytics/rest-client/python'`.
 - (e) Choose *classes*.
 - (f) Choose *myproject* and click *OK* button.
 - (g) Name the library "ta-python-client".
 - (h) Choose *OK* button.
4. Choose *File* → *New* → *Python File*.
5. Name the file 'test' and type in the following code:

⁴<http://www.jetbrains.com/idea/>


```
import trustedanalytics as ta
ta.server.host = "correct host name or IP address"
ta.connect()
ta.server.ping()
```

6. Choose *Run* → *Run*, you should see the output:

```
Successful ping to Trusted Analytics ATK at http://localhost:9099/info
```

7. Next take a look at the included examples.

21.3 PyCharm Setup Instructions

PyCharm is a Python IDE created by JetBrains.

Table of Contents

- *Prerequisites*
- *Setup*

21.3.1 Prerequisites

These instructions assume you have already installed:

- Python 2.7.
- Trusted Analytics Python REST Client and required dependencies.
- PyCharm⁵.

Trusted Analytics should work with any version of PyCharm but these instructions were tested with PyCharm Community Edition 3.4.1.

21.3.2 Setup

1. Select *New Project* on PyCharm's initial screen.
 - (a) Give your project a name, for example, "myproject".
 - (b) Choose *Python 2.7* as the Python Interpreter and choose the *OK* button.
 - i. If *Python 2.7* does not appear in the list you will need to configure a Python 2.7 Interpreter.
 - A. Choose the button that looks like a "gear".
 - B. Choose *Add Local*.
 - C. Browse for your local Python 2.7 installation. On RedHat or Centos this is probably `/usr/bin/python`.
 - D. Choose the *OK* button.
2. Choose *File* → *Settings*.
 - (a) Choose *Project Structure*.

⁵<http://www.jetbrains.com/pycharm/>

- (b) Choose *Add Content Root* and browse to the Trusted Analytics Python REST Client libraries. On RedHat or Centos these are found under `/usr/lib/trustedanalytics/rest-client/python`.
 - (c) Choose *Apply* button.
 - (d) Choose *OK* button.
3. Right click your project folder, for example, “myproject”, and select *New* → *Python File*.
- (a) Name the file “test” and type in the following code:

```
import trustedanalytics as ta
ta.server.host = "correct host name or IP address"
ta.connect()
ta.server.ping()
```

- (b) If you see a yellow bar across the top of the file warning about “Package requirements” not being satisfied then your system is not setup correctly.
 - i. You may not have installed all of the Python dependencies for the Trusted Analytics REST Client correctly.
 - ii. You may have chosen the wrong Python interpreter.
4. Choose *Run* → *Run*, you should see the output:

```
Successful ping to Trusted Analytics ATK at http://localhost:9099/info
```

5. Next take a look at the included examples.

Part V

Python API

CONNECT TO THE SERVER

Table of Contents

- *Basic connecting*
- *Connections requiring OAuth*
- *Using Environmental Variables*
- *Troubleshooting*
 - *Client's Server Settings*

The Python client must 'connect' to an Trusted Analytics server before it can be used. Here is the 'connect' process described by the method's documentation:

```
trustedanalytics.connect (self, credentials_file=None)
```

Connect to the trustedanalytics server.

This method calls the server, downloads its API information and dynamically generates and adds the appropriate Python code to the Python package for this python session. Calling this method is required before invoking any server activity.

After the client has connected to the server, the server config cannot be changed. User must restart Python in order to change connection info.

Subsequent calls to this method invoke no action.

There is no "connection" object or notion of being continuously "connected". The call to connect is just a one-time process to download the API and prepare the client. If the server goes down and comes back up, this client will not recognize any difference from a connection point of view, and will still be operating with the API information originally downloaded.

Parameters **credentials_file**: str (optional)

file name of a credentials file. If supplied, it will override the settings authentication settings in the client's server configuration. The credentials file is normally obtained through the env.

22.1 Basic connecting

To use the default settings provided by the environment and/or configuration:

```
>>> import trustedanalytics as ta
>>> ta.connect()
```

To connect to a specific server:

```
>>> import trustedanalytics as ta
>>> ta.server.uri = 'myhost-name:port'
>>> ta.connect()
```

22.2 Connections requiring OAuth

To connect to a DP2 instance of Trusted Analytics, the python client must have an OAuth access token (see [oauth tokens](<http://self-issued.info/docs/draft-ietf-oauth-v2-bearer.html>)). The user must have a credentials file which holds an OAuth access token and a refresh token.

The user can create a credentials file using Trusted Analytics client running in an interactive python session. Call `create_credentials_file('filename_of_your_choice')` and interactively provide answers to its prompt.

```
$ python2.7

>>> import trustedanalytics as ta
>>> ta.create_connect_file('~/.ta/demo.creds')
OAuth server URI: uaa.my-dp2-domain.com
user name: dscientist9
Password: *****

Credentials created at '/home/dscientist9/.ta/demo.creds'
```

The credentials file can be specified when calling `connect` or set as an environmental variable `$TA_CREDS`.

```
>>> ta.connect('~/.ta/demo.creds')
Connected. This client instance connected to
server http://my-ta-instance.my-dp2-apps-domain.com/v1
as user dscientist9 at 2015-06-19 10:27:21.583704.
```

The credentials file path must be relative to how python was launched. Full paths are recommended. Multiple credentials files can be created. They should be protected with appropriate OS privileges.

22.3 Using Environmental Variables

The URI of the Trusted Analytics server can be specified by the environmental variable `$TA_URI`. The python client will initialize its config setting to this value. It may still be overridden as shown above in the session or script.

```
$ export TA_URI=ta-server.demo-gotapaas.com
```

The credentials file can be specified by `$TA_CREDS`.

```
$ export TA_CREDS=~/.ta/demo.creds
```

With these two variables set, the simple connect sequence works.

```
>>> import trustedanalytics as ta
>>> ta.connect()
```


DATA TYPES

All data manipulated and stored using frames and graphs must fit into one of the supported Python data types.

```
>>> ta.valid_data_types

float32, float64, ignore, int32, int64, unicode, vector(n), datetime
(and aliases: float->float64, int->int32, list->vector, long->int64, str->unicode)
```

date-time	[ALPHA] (This is a function or feature which has been developed, but has not been completely tested. Use this function with caution. This function may be changed or eliminated in future releases.) object for date and time; equivalent to python's datetime.datetime class. Converts to and from strings using the ISO 8601 format. Inside the server, the object is represented by the nscala/joda DateTime object. When interfacing with various data sources and sinks that use different data types for datetime, the datetime value will be converted to a string by default.
float32	32-bit floating point number; equivalent to numpy.float32
float64	64-bit floating point number; equivalent to numpy.float64
ig-nore	type available to describe a field in a data source that the parser should ignore
int32	32-bit integer; equivalent to numpy.int32
int64	32-bit integer; equivalent to numpy.int64
uni-code	Python's unicode representation for strings.
vec-tor(n)	[ALPHA] Ordered list of n float64 numbers (array of fixed-length n); uses numpy.ndarray

Note: Numpy values of positive infinity (np.inf), negative infinity (-np.inf) or nan (np.nan) are treated as Python's None when sent to the server. Results of any user-defined functions which deal with such values are automatically converted to None. Any further usage of those data points should treat the values as None.

API Maturity Tags

Functions in the API may be at different levels of software maturity. Where a function is not mature, the documentation will note it with one of the following tags. The absence of a tag means the function is standardized and fully tested.

[ALPHA]

[BETA] (This is a function or feature which has been developed and preliminarily tested, but has not been completely tested. Use this function with caution. This function may be changed in future releases.)

[DEPRECATED] (This is a function or feature which is no longer supported. It is recommended that an alternate solution be found. This function may be removed in future releases.)

DATA SOURCES

Table of Contents

- *CsvFile*
- *HiveQuery*
- *HBase*
- *Jdbc*
- *JsonFile*
- *LineFile*
- *Pandas*
- *XmlFile*

24.1 CsvFile

class `trustedanalytics.CsvFile` (*file_name*, *schema*, *delimiter*=' ', *skip_header_lines*=0)
Define a CSV file.

Attributes

<i>field_names</i>	Schema field names from the <code>CsvFile</code> class.
<i>field_types</i>	Schema field types from the <code>CsvFile</code> class.

__init__ (*file_name*, *schema*, *delimiter*=' ', *skip_header_lines*=0)
Define a CSV file.

Parameters *file_name* : str

The name of the file containing data in a CSV format. The file must be in the Hadoop file system. Relative paths are interpreted as being relative to the path set in the application configuration file. See `Configure File System Root`. Absolute paths (beginning with `hdfs://...`, for example) are also supported.

schema : list of tuples of the form (string, type)

A description of the fields of data in the form of a list of tuples, which describe each field. Each tuple is in the form (name, type), where the name is a string, and type is a supported data type. Upon import of the data, the name becomes the name of a column, so the names must be unique and follow column naming rules. For a list of

valid data types, see [Data Types](#). The type `ignore` may also be used if the field should be ignored on loads.

delimiter : str (optional)

A string which indicates the separation of the data fields. This is usually a single character and could be a non-visible character such as a tab. This string must be enclosed by quotes in the command declaration, for example `" , "`.

skip_header_lines : int (optional)

An integer for the numbers of lines to skip before parsing records.

Returns class

A class which holds both the name and schema of a CSV file.

Notes

Unicode characters should not be used in the column name, because some functions do not support them and will not operate properly.

Examples

Given a raw data file named `'raw_data.csv'`, located at `'hdfs://localhost.localdomain/user/trusted/data/'`. It consists of three columns, *a*, *b*, and *c*. The columns have the data types *int32*, *int32*, and *str* respectively. The fields of data are separated by commas. There is no header to the file.

Import the Trusted Analytics:

```
>>> import trustedanalytics as ta
```

Define the data:

```
>>> csv_schema = [("a", ta.int32), ("b", ta.int32), ("c", str)]
```

Create a `CsvFile` object with this schema:

```
>>> csv_define = ta.CsvFile("data/raw_data.csv", csv_schema)
```

The default delimiter, a comma, was used to separate fields in the file, so it was not specified. If the columns of data were separated by a character other than comma, the appropriate delimiter would be specified. For example if the data columns were separated by the colon character, the instruction would be:

```
>>> ta.CsvFile("data/raw_data.csv", csv_schema, delimiter = ':')
```

If the data had some lines of header at the beginning of the file, the lines should be skipped:

```
>>> csv_data = ta.CsvFile("data/raw_data.csv", csv_schema, skip_header_lines=2)
```

For other examples see [Importing a CSV File](#).

field_names

Schema field names from the `CsvFile` class.

Returns list

A list of field name strings.

Examples

Given a raw data file 'raw_data.csv' with columns *col1* (*int32*) and *col2* (*float32*):

```
>>> csv_class = ta.CsvFile("raw_data.csv",
... schema=[("col1", ta.int32), ("col2", ta.float32)])
>>> print(csv_class.field_names())
```

Results:

```
["col1", "col2"]
```

field_types

Schema field types from the CsvFile class.

Returns list

A list of field types.

Examples

Given a raw data file 'raw_data.csv' with columns *col1* (*int32*) and *col2* (*float32*):

Results:

```
[ta.int32, ta.float32]
```

24.2 HiveQuery

class `trustedanalytics.HiveQuery` (*query*)

Define the sql query to retrieve the data from a Hive table.

Methods

__init__ (*query*)

Define the sql query to retrieve the data from a Hive table.

Only a subset of Hive data types are supported.

Data Type	Support
boolean	cast to int
bigint	native support
int	native support
tinyint	cast to int
smallint	cast to int
decimal	cast to double, may lose precision
double	native support
float	native support

date	cast to string
string	native support
timestamp	cast to string
varchar	cast to string
arrays	not supported
binary	not supported
char	not supported
maps	not supported
structs	not supported
union	not supported

Parameters `query` : str

The sql query to retrieve the data

Returns `class` : HiveQuery object

An object which holds Hive sql query.

Examples

Given a Hive table *person* having *name* and *age* among other columns. A simple query could be to get the query for the name and age .. code:

```
>>> import trustedanalytics as ta
>>> ta.connect()
```

Define the data:

```
>>> hive_query = ta.HiveQuery("select name, age from person")
```

Create a frame using the object:

```
>>> my_frame = ta.Frame(hive_query)
```

24.3 HBase

class `trustedanalytics.HBaseTable` (*table_name*, *schema*, *start_row=None*, *end_row=None*)

Define the object to retrieve the data from an hBase table.

Methods

__init__ (*table_name*, *schema*, *start_row=None*, *end_row=None*)

Define the object to retrieve the data from an hBase table.

Parameters `my_table` : str

The table name

schema : List of (column family, column name, data type for the cell value)

Returns `class` : HBaseTable object

An object which holds hBase data.

Examples

```
>>> import trustedanalytics as ta
>>> ta.connect()
>>> h = tp.HBaseTable ("my_table", [("pants", "aisle", unicode), ("pants", "row", int), ("shirts", "size", int)])
>>> f = tp.Frame(h)
>>> f.inspect()
```

24.4 Jdbc

class `trustedanalytics.JdbcTable` (*table_name*, *connector_type=None*, *url=None*, *driver_name=None*, *query=None*)

Define the object to retrieve the data from an jdbc table.

Methods

__init__ (*table_name*, *connector_type=None*, *url=None*, *driver_name=None*, *query=None*)

Define the object to retrieve the data from an jdbc table.

Parameters *table_name* : str

the table name

connector_type : str

the connector type

url : str

Jdbc connection string (as url)

driver_name : str

An optional driver name

query : initial query (for data filtering / processing)

Returns *class* : JdbcTable object

An object which holds jdbc data.

Examples

```
>>> import trustedanalytics as ta
>>> ta.connect()
>>> jdbcTable = tp.JdbcTable ("test",
                                "jdbc:sqlserver://localhost/SQLExpress;datasource=somedatabase;
                                "com.microsoft.sqlserver.jdbc.SQLServerDriver",
                                "select * FROM SomeTable")
>>> frame = tp.Frame(jdbcTable)
>>> frame.inspect()
```

24.5 JsonFile

class `trustedanalytics.JsonFile` (*file_name*)

Define a file as having data in JSON format.

__init__ (*file_name*)

Define a file as having data in JSON format. When JSON files are loaded into the system all top level JSON objects are recorded into the frame as separate elements.

Parameters *file_name* : str

Name of data input file. File must be in the Hadoop file system. Relative paths are interpreted relative to the `trustedanalytics.atk.engine.fs.root` configuration. Absolute paths (beginning with `hdfs://...`, for example) are also supported. See Configure File System Root.

Returns class

An object which holds both the name and tag of a JSON file.

Examples

Given a raw data file named 'raw_data.json' located at 'hdfs://localhost.localdomain/user/trusted/data/'. It consists of a 3 top level json objects with a single value each called obj. Each object contains the attributes color, size, and shape.

The example JSON file:

```
{ "obj": {
  "color": "blue",
  "size": 3,
  "shape": "square" }
}
{ "obj": {
  "color": "green",
  "size": 7,
  "shape": "triangle" }
}
{ "obj": {
  "color": "orange",
  "size": 10,
  "shape": "square" }
}
```

Import the Trusted Analytics:

```
>>> import trustedanalytics as ta
>>> ta.connect()
```

Define the data:

```
>>> json_file = ta.JsonFile("data/raw_data.json")
```

Create a frame using this JsonFile:

```
>>> my_frame = ta.Frame(json_file)
```

The frame looks like:


```

data_lines
/-----/
'{"obj": {
  "color": "blue",
  "size": 3,
  "shape": "square" }}'
'{"obj": {
  "color": "green",
  "size": 7,
  "shape": "triangle" }}'
'{"obj": {
  "color": "orange",
  "size": 10,
  "shape": "square" }}'

```

Parse values out of the XML column using the `add_columns` method:

```

>>> def parse_my_json(row):
...     import json
...     my_json = json.loads(row[0])
...     obj = my_json['obj']
...     return (obj['color'], obj['size'], obj['shape'])

>>> my_frame.add_columns(parse_my_json, ["color", str], ["size", str],
... ("shape", str))

```

Original XML column is no longer necessary:

```

>>> my_frame.drop_columns(['data_lines'])

```

Result:

```

>>> my_frame.inspect()

  color:str  size:str  shape:str
/-----/
blue        3        square
green       7        triangle
orange     10        square

```

24.6 LineFile

class `trustedanalytics.LineFile` (*file_name*)

Define a line-separated file.

__init__ (*file_name*)

Define a line-separated file.

Parameters `file_name` : str

Name of data input file. File must be in the Hadoop file system. Relative paths are interpreted relative to the `trustedanalytics.atk.engine.fs.root` configuration. Absolute paths (beginning with `hdfs://...`, for example) are also supported. See [Configure File System Root](#).

Returns class

A class which holds the name of a Line File.

Examples

Given a raw data file ‘rawline_data.txt’ located at ‘hdfs://localhost.localdomain/user/trusted/data/’. It consists of multiple lines separated by new line character.

Import the Trusted Analytics:

```
>>> import trustedanalytics as ta
>>> ta.connect()
```

Define the data:

```
>>> linefile_class = ta.LineFile("data/rawline_data.txt")
```

24.7 Pandas

class `trustedanalytics.Pandas` (*pandas_frame*, *schema*, *row_index=True*)

Defines a pandas data source

Attributes

<i>field_names</i>	Schema field names.
<i>field_types</i>	Schema field types

__init__ (*pandas_frame*, *schema*, *row_index=True*)

Defines a pandas data source

Parameters **pandas_frame** : a pandas dataframe object

schema : list of tuples of the form (string, type)

schema description of the fields for a given line. It is a list of tuples which describe each field, (field name, field type), where the field name is a string, and file is a supported type, (See *data_types* from the *atktypes* module). Unicode characters should not be used in the column name.

row_index : boolean (optional)

indicates if the *row_index* is present in the pandas dataframe and needs to be ignored when looking at the data values. Default value is True.

Returns class

An object which holds both the pandas dataframe and schema associated with it.

Examples

For this example, we are going to use a raw data file named “pandas_df.csv”. It consists of three columns named: *a*, *b*, *c*. The columns have the data types: *int32*, *int32*, *str*. The fields of data are separated by commas. ‘0th’ row in the file indicates the header.

First bring in the stuff:

```
import trustedanalytics as ta
import pandas
```

At this point create a schema that defines the data:

```
schema = [("a", ta.int32),
          ("b", ta.int32),
          ("c", str)]
```

`your_pandas = pandas.read_csv("pandas_df.csv")`

Now build a `PandasFrame` object with this schema:

```
my_pandas = ta.PandasFrame(your_pandas, schema, False)
```

field_names

Schema field names.

List of field names from the schema stored in the `trustedanalytics pandas dataframe object`

Returns list of string

Field names

Examples

For this example, we are going to use a pandas dataframe object `your_pandas`. It will have two columns `col1` and `col2` with types of `int32` and `float32` respectively:

```
my_pandas = ta.PandasFrame(your_pandas, schema=[("col1", ta.int32), ("col2", ta.float32)])
print(my_pandas.field_names())
```

The output would be:

```
["col1", "col2"]
```

field_types

Schema field types

List of field types from the schema stored in the `trustedanalytics pandas dataframe object`.

Returns list of types

Field types

Examples

For this example, we are going to use a pandas dataframe object `your_pandas`. It will have two columns `col1` and `col2` with types of `int32` and `float32` respectively:

```
my_pandas = ta.PandasFrame(your_pandas, schema=[("col1", ta.int32), ("col2", ta.float32)])
print(my_csv.field_types())
```

The output would be:

```
[numpy.int32, numpy.float32]
```

24.8 XmlFile

class `trustedanalytics.XmlFile` (*file_name*, *tag_name*)

Define an file as having data in XML format.

__init__ (*file_name*, *tag_name*)

Define an file as having data in XML format.

When XML files are loaded into the system, individual records are separated into the highest level elements found with the specified tag name and placed into a column called `data_lines`.

Parameters `file_name` : str

Name of data input file. File must be in the Hadoop file system. Relative paths are interpreted relative to the `trustedanalytics.atk.engine.fs.root` configuration. Absolute paths (beginning with `hdfs://...`, for example) are also supported. See Configure File System Root.

tag_name : str

Tag name used to determine the split of elements into separate records.

Returns class

An object which holds both the name and tag of a XML file.

Examples

Given a raw data file named 'raw_data.xml' located at 'hdfs://localhost.localdomain/user/trusted/data/'. It consists of a root element called *shapes* with subelements with the tag names *square* and *triangle*. Each of these subelements has two potential subelements called *name* and *size*. One of the elements has an attribute called *color*. Additionally, the subelement *triangle* is not needed so we can skip it during the import.

The example XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<shapes>
  <square>
    <name>left</name>
    <size>3</size>
  </square>
  <triangle>
    <size>3</size>
  </triangle>
  <square color="blue">
    <name>right</name>
    <size>5</size>
  </square>
</shapes>
```

Import the Trusted Analytics:

```
>>> import trustedanalytics as ta
>>> ta.connect()
```

Define the data:

```
>>> xml_file = ta.XmlFile("data/raw_data.xml", "square")
```

Create a frame using this XmlFile:

```
>>> my_frame = ta.Frame(xml_file)
```

The frame looks like:

```
data_lines
/-----/
'<square>
  <name>left</name>
  <size>3</size>
</square>'
'<square color="blue">
  <name>right</name>
  <size>5</size>
</square>'
```

Parse values out of the XML column using the add_columns method:

```
>>> def parse_my_xml(row):
...     import xml.etree.ElementTree as ET
...     ele = ET.fromstring(row[0])
...     return (ele.get("color"), ele.find("name").text, ele.find("size").text)

>>> my_frame.add_columns(parse_my_xml, [("color", str), ("name", str), ("size", str)])
```

Original XML column is no longer necessary:

```
>>> my_frame.drop_columns(['data_lines'])
```

Result:

```
>>> my_frame.inspect()

color:str    name:str    size:str
/-----/
None        left        3
blue        right        5
```


Classes

25.1 *Frames* EdgeFrame

25.1.1 *EdgeFrame* `__init__`

`__init__(self, graph=None, label=None, src_vertex_label=None, dest_vertex_label=None, directed=None)`

Examples

Parameters `graph` : ? (default=None)

graph these edges belong to

label : ? (default=None)

edge label

src_vertex_label : ? (default=None)

label of the source vertex type

dest_vertex_label : ? (default=None)

label of the destination vertex type

directed : ? (default=None)

directed or undirected

Returns : VertexFrame object

An object with access to the frame.

Given a data file `/movie.csv`, create a frame to match this data and move the data to the frame. Create an empty graph and define some vertex and edge types.

```
>>> my_csv = ta.CsvFile("/movie.csv", schema= [('user_id', int32),
...                                           ('user_name', str),
...                                           ('movie_id', int32),
...                                           ('movie_title', str),
...                                           ('rating', str)])
```

```
>>> my_frame = ta.Frame(my_csv)
>>> my_graph = ta.Graph()
>>> my_graph.define_vertex_type('users')
>>> my_graph.define_vertex_type('movies')
>>> my_graph.define_edge_type('ratings', 'users', 'movies', directed=True)
```

Add data to the graph from the frame:

```
>>> my_graph.vertices['users'].add_vertices(my_frame, 'user_id',
... ['user_name'])
>>> my_graph.vertices['movies'].add_vertices(my_frame, 'movie_id', ['movie_title'])
```

Create an edge frame from the graph, and add edge data from the frame.

```
>>> my_edge_frame = graph.edges['ratings']
>>> my_edge_frame.add_edges(my_frame, 'user_id', 'movie_id', ['rating'])
```

Retrieve a previously defined graph and retrieve an EdgeFrame from it:

```
>>> my_old_graph = ta.get_graph("your_graph")
>>> my_new_edge_frame = my_old_graph.edges["your_label"]
```

Calling methods on an EdgeFrame:

```
>>> my_new_edge_frame.inspect(20)
```

Copy an EdgeFrame to a frame using the copy method:

```
>>> my_new_frame = my_new_edge_frame.copy()
```

25.1.2 EdgeFrame add_columns

add_columns (*self, func, schema, columns_accessed=None*)

Add columns to current frame.

Parameters **func** : UDF

User-Defined Function (UDF) which takes the values in the row and produces a value, or collection of values, for the new cell(s).

schema : tuple | list of tuples

The schema for the results of the UDF, indicating the new column(s) to add. Each tuple provides the column name and data type, and is of the form (str, type).

columns_accessed : list (default=None)

List of columns which the UDF will access. This adds significant performance benefit if we know which column(s) will be needed to execute the UDF, especially when the frame has significantly more columns than those being used to evaluate the UDF.

Assigns data to column based on evaluating a function for each row.

Notes

1. The row UDF ('func') must return a value in the same format as specified by the schema. See [Python User Functions](#).

2.Unicode in column names is not supported and will likely cause the `drop_frames()` method (and others) to fail!

Examples

Given a Frame *my_frame* identifying a data frame with two int32 columns *column1* and *column2*. Add a third column *column3* as an int32 and fill it with the contents of *column1* and *column2* multiplied together:

```
>>> my_frame.add_columns(lambda row: row.column1*row.column2,
... ('column3', int32))
```

The frame now has three columns, *column1*, *column2* and *column3*. The type of *column3* is an int32, and the value is the product of *column1* and *column2*.

Add a string column *column4* that is empty:

```
>>> my_frame.add_columns(lambda row: '', ('column4', str))
```

The Frame object *my_frame* now has four columns *column1*, *column2*, *column3*, and *column4*. The first three columns are int32 and the fourth column is str. Column *column4* has an empty string ("") in every row.

Multiple columns can be added at the same time. Add a column *a_times_b* and fill it with the contents of column *a* multiplied by the contents of column *b*. At the same time, add a column *a_plus_b* and fill it with the contents of column *a* plus the contents of column *b*:

```
>>> my_frame.add_columns(lambda row: [row.a * row.b, row.a +
... row.b], [("a_times_b", float32), ("a_plus_b", float32)])
```

Two new columns are created, "a_times_b" and "a_plus_b", with the appropriate contents.

Given a frame of data and Frame *my_frame* points to it. In addition we have defined a UDF *func*. Run *func* on each row of the frame and put the result in a new int column *calculated_a*:

```
>>> my_frame.add_columns(func, ("calculated_a", int))
```

Now the frame has a column *calculated_a* which has been filled with the results of the UDF *func*.

A UDF must return a value in the same format as the column is defined. In most cases this is automatically the case, but sometimes it is less obvious. Given a UDF *function_b* which returns a value in a list, store the result in a new column *calculated_b*:

```
>>> my_frame.add_columns(function_b, ("calculated_b", float32))
```

This would result in an error because *function_b* is returning a value as a single element list like [2.4], but our column is defined as a tuple. The column must be defined as a list:

```
>>> my_frame.add_columns(function_b, [("calculated_b", float32)])
```

To run an optimized version of `add_columns`, `columns_accessed` parameter can be populated with the column names which are being accessed in UDF. This speeds up the execution by working on only the limited feature set than the entire row.

Let's say a frame has 4 columns named *a*, *b*, *c* and *d* and we want to add a new column with value from column *a* multiplied by value in column *b* and call it *a_times_b*. In the example below, `columns_accessed` is a list with column names *a* and *b*.

```
>>> my_frame.add_columns(lambda row: row.a * row.b, ("a_times_b", float32), columns_accessed=["a", "b"])
```

`add_columns` would fail if `columns_accessed` parameter is not populated with the correct list of accessed columns. If not specified, `columns_accessed` defaults to `None` which implies that all columns might be accessed by the UDF.

More information on a row UDF can be found at [Python User Functions](#)

25.1.3 *EdgeFrame* `add_edges`

`add_edges` (*self*, *source_frame*, *column_name_for_source_vertex_id*, *column_name_for_dest_vertex_id*, *column_names=None*, *create_missing_vertices=False*)

Add edges to a graph.

Parameters **`source_frame`** : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

Frame that will be the source of the edge data.

`column_name_for_source_vertex_id` : unicode

column name for a unique id for each source vertex (this is not the system defined `_vid`).

`column_name_for_dest_vertex_id` : unicode

column name for a unique id for each destination vertex (this is not the system defined `_vid`).

`column_names` : list (default=`None`)

Column names to be used as properties for each vertex, `None` means use all columns, empty list means use none.

`create_missing_vertices` : bool (default=`False`)

True to create missing vertices for edge (slightly slower), False to drop edges pointing to missing vertices. Defaults to False.

Returns : `_Unit`

Includes appending to a list of existing edges.

25.1.4 *EdgeFrame* `assign_sample`

`assign_sample` (*self*, *sample_percentages*, *sample_labels=None*, *output_column=None*, *random_seed=None*)

Randomly group rows into user-defined classes.

Parameters **`sample_percentages`** : list

Entries are non-negative and sum to 1. (See the note below.) If the i 'th entry of the list is p , then each row receives label i with independent probability p .

`sample_labels` : list (default=`None`)

Names to be used for the split classes. Defaults "TR", "TE", "VA" when the length of *sample_percentages* is 3, and defaults to `Sample_0`, `Sample_1`, ... otherwise.

`output_column` : unicode (default=`None`)

Name of the new column which holds the labels generated by the function.

random_seed : int32 (default=None)

Random seed used to generate the labels. Defaults to 0.

Returns : _Unit

Randomly assign classes to rows given a vector of percentages. The table receives an additional column that contains a random label. The random label is generated by a probability distribution function. The distribution function is specified by the `sample_percentages`, a list of floating point values, which add up to 1. The labels are non-negative integers drawn from the range $[0, \text{len}(S) - 1]$ where S is the `sample_percentages`. Optionally, the user can specify a list of strings to be used as the labels. If the number of labels is 3, the labels will default to “TR”, “TE” and “VA”.

Notes

The sample percentages provided by the user are preserved to at least eight decimal places, but beyond this there may be small changes due to floating point imprecision.

In particular:

1. The engine validates that the sum of probabilities sums to 1.0 within eight decimal places and returns an error if the sum falls outside of this range.
2. The probability of the final class is clamped so that each row receives a valid label with probability one.

25.1.5 EdgeFrame bin_column

bin_column (*self*, *column_name*, *cutoffs*, *include_lowest=None*, *strict_binning=None*,
bin_column_name=None)
 Classify data into user-defined groups.

Parameters **column_name** : unicode

Name of the column to bin.

cutoffs : list

Array of values containing bin cutoff points. Array can be list or tuple. Array values must be progressively increasing. All bin boundaries must be included, so, with N bins, you need $N+1$ values.

include_lowest : bool (default=None)

Specify how the boundary conditions are handled. True indicates that the lower bound of the bin is inclusive. False indicates that the upper bound is inclusive. Default is True.

strict_binning : bool (default=None)

Specify how values outside of the cutoffs array should be binned. If set to True, each value less than `cutoffs[0]` or greater than `cutoffs[-1]` will be assigned a bin value of -1. If set to False, values less than `cutoffs[0]` will be included in the first bin while values greater than `cutoffs[-1]` will be included in the final bin.

bin_column_name : unicode (default=None)

The name for the new binned column. Default is `<column_name>_binned`.

Returns : _Unit

Summarize rows of data based on the value in a single column by sorting them into bins, or groups, based on a list of bin cutoff points.

Notes

1. Unicode in column names is not supported and will likely cause the `drop_frames()` method (and others) to fail!
2. Bins IDs are 0-index: the lowest bin number is 0.
3. The first and last cutoffs are always included in the bins. When `include_lowest` is `True`, the last bin includes both cutoffs. When `include_lowest` is `False`, the first bin (bin 0) includes both cutoffs.

25.1.6 *EdgeFrame* `bin_column_equal_depth`

`bin_column_equal_depth` (*self*, *column_name*, *num_bins=None*, *bin_column_name=None*)

Classify column into groups with the same frequency.

Parameters `column_name` : unicode

The column whose values are to be binned.

`num_bins` : int32 (default=None)

The maximum number of bins. Default is the Square-root choice $\lfloor \sqrt{m} \rfloor$, where m is the number of rows.

`bin_column_name` : unicode (default=None)

The name for the new column holding the grouping labels. Default is `<column_name>_binned`.

Returns : dict

A list containing the edges of each bin.

Group rows of data based on the value in a single column and add a label to identify grouping.

Equal depth binning attempts to label rows such that each bin contains the same number of elements. For n bins of a column C of length m , the bin number is determined by:

$$\lceil n * \frac{f(C)}{m} \rceil$$

where f is a tie-adjusted ranking function over values of C . If there are multiples of the same value in C , then their tie-adjusted rank is the average of their ordered rank values.

Notes

1. Unicode in column names is not supported and will likely cause the `drop_frames()` method (and others) to fail!
2. The `num_bins` parameter is considered to be the maximum permissible number of bins because the data may dictate fewer bins. For example, if the column to be binned has a quantity of X elements with only 2 distinct values and the `num_bins` parameter is greater than 2, then the actual number of bins will only be 2. This is due to a restriction that elements with an identical value must belong to the same bin.

25.1.7 *EdgeFrame* bin_column_equal_width

bin_column_equal_width (*self*, *column_name*, *num_bins=None*, *bin_column_name=None*)

Classify column into same-width groups.

Parameters **column_name** : unicode

The column whose values are to be binned.

num_bins : int32 (default=None)

The maximum number of bins. Default is the Square-root choice $\lfloor \sqrt{m} \rfloor$, where m is the number of rows.

bin_column_name : unicode (default=None)

The name for the new column holding the grouping labels. Default is `<column_name>_binned`.

Returns : dict

A list of the edges of each bin.

Group rows of data based on the value in a single column and add a label to identify grouping.

Equal width binning places column values into groups such that the values in each group fall within the same interval and the interval width for each group is equal.

Notes

1. Unicode in column names is not supported and will likely cause the `drop_frames()` method (and others) to fail!
2. The `num_bins` parameter is considered to be the maximum permissible number of bins because the data may dictate fewer bins. For example, if the column to be binned has 10 elements with only 2 distinct values and the `num_bins` parameter is greater than 2, then the number of actual number of bins will only be 2. This is due to a restriction that elements with an identical value must belong to the same bin.

25.1.8 *EdgeFrame* categorical_summary

categorical_summary (*self*, *column_inputs=None*)

[ALPHA] Compute a summary of the data in a column(s) for categorical or numerical data types.

Parameters **column_inputs** : str | tuple(str, dict) (default=None)

Comma-separated column names to summarize or tuple containing column name and dictionary of optional parameters. Optional parameters (see below for details): `top_k` (default = 10), `threshold` (default = 0.0)

Returns : dict

Summary for specified column(s) consisting of levels with their frequency and percentage

The returned value is a Map containing categorical summary for each specified column.

For each column, levels which satisfy the top k and/or threshold cutoffs are displayed along with their frequency and percentage occurrence with respect to the total rows in the dataset.

Missing data is reported when a column value is empty ("") or null.

All remaining data is grouped together in the Other category and its frequency and percentage are reported as well.

User must specify the column name and can optionally specify top_k and/or threshold.

Optional parameters:

top_k Displays levels which are in the top k most frequently occurring values for that column.

threshold Displays levels which are above the threshold percentage with respect to the total row count.

top_k and threshold Performs level pruning first based on top k and then filters out levels which satisfy the threshold criterion.

defaults Displays all levels which are in Top 10.

Examples

```
>>> frame.categorical_summary('source', 'target')
>>> frame.categorical_summary(('source', {'top_k' : 2}))
>>> frame.categorical_summary(('source', {'threshold' : 0.5}))
>>> frame.categorical_summary(('source', {'top_k' : 2}), ('target',
... {'threshold' : 0.5}))
```

Sample output (for last example above):

```
>>> {u'categorical_summary': [{u'column': u'source', u'levels': [
... {u'percentage': 0.32142857142857145, u'frequency': 9, u'level': u'thing'},
... {u'percentage': 0.32142857142857145, u'frequency': 9, u'level': u'abstraction'},
... {u'percentage': 0.25, u'frequency': 7, u'level': u'physical_entity'},
... {u'percentage': 0.10714285714285714, u'frequency': 3, u'level': u'entity'},
... {u'percentage': 0.0, u'frequency': 0, u'level': u'Missing'},
... {u'percentage': 0.0, u'frequency': 0, u'level': u'Other'}]],
... {u'column': u'target', u'levels': [
... {u'percentage': 0.07142857142857142, u'frequency': 2, u'level': u'thing'},
... {u'percentage': 0.07142857142857142, u'frequency': 2,
... u'level': u'physical_entity'},
... {u'percentage': 0.07142857142857142, u'frequency': 2, u'level': u'entity'},
... {u'percentage': 0.03571428571428571, u'frequency': 1, u'level': u'variable'},
... {u'percentage': 0.03571428571428571, u'frequency': 1, u'level': u'unit'},
... {u'percentage': 0.03571428571428571, u'frequency': 1, u'level': u'substance'},
... {u'percentage': 0.03571428571428571, u'frequency': 1, u'level': u'subject'},
... {u'percentage': 0.03571428571428571, u'frequency': 1, u'level': u'set'},
... {u'percentage': 0.03571428571428571, u'frequency': 1, u'level': u'reservoir'},
... {u'percentage': 0.03571428571428571, u'frequency': 1, u'level': u'relation'},
... {u'percentage': 0.0, u'frequency': 0, u'level': u'Missing'},
... {u'percentage': 0.5357142857142857, u'frequency': 15, u'level': u'Other'}]]}]}
```

25.1.9 EdgeFrame classification_metrics

classification_metrics (*self*, *label_column*, *pred_column*, *pos_label*=None, *beta*=None)

Model statistics of accuracy, precision, and others.

Parameters *label_column* : unicode

The name of the column containing the correct label for each instance.

pred_column : unicode

The name of the column containing the predicted label for each instance.

pos_label : None (default=None)

beta : float64 (default=None)

This is the beta value to use for F_β measure (default F1 measure is computed); must be greater than zero. Defaults is 1.

Returns : dict

object <object>.accuracy : double <object>.confusion_matrix : table
 <object>.f_measure : double <object>.precision : double <object>.recall : double

Calculate the accuracy, precision, confusion_matrix, recall and F_β measure for a classification model.

- The **f_measure** result is the F_β measure for a classification model. The F_β measure of a binary classification model is the harmonic mean of precision and recall. If we let:

–beta $\equiv \beta$,

– T_P denotes the number of true positives,

– F_P denotes the number of false positives, and

– F_N denotes the number of false negatives

then:

$$F_\beta = (1 + \beta^2) * \frac{\frac{T_P}{T_P + F_P} * \frac{T_P}{T_P + F_N}}{\beta^2 * \frac{T_P}{T_P + F_P} + \frac{T_P}{T_P + F_N}}$$

The F_β measure for a multi-class classification model is computed as the weighted average of the F_β measure for each label, where the weight is the number of instances of each label. The determination of binary vs. multi-class is automatically inferred from the data.

- The **recall** result of a binary classification model is the proportion of positive instances that are correctly identified. If we let T_P denote the number of true positives and F_N denote the number of false negatives, then the model recall is given by $\frac{T_P}{T_P + F_N}$.

For multi-class classification models, the recall measure is computed as the weighted average of the recall for each label, where the weight is the number of instances of each label. The determination of binary vs. multi-class is automatically inferred from the data.

- The **precision** of a binary classification model is the proportion of predicted positive instances that are correctly identified. If we let T_P denote the number of true positives and F_P denote the number of false positives, then the model precision is given by: $\frac{T_P}{T_P + F_P}$.

For multi-class classification models, the precision measure is computed as the weighted average of the precision for each label, where the weight is the number of instances of each label. The determination of binary vs. multi-class is automatically inferred from the data.

- The **accuracy** of a classification model is the proportion of predictions that are correctly identified. If we let T_P denote the number of true positives, T_N denote the number of true negatives, and K denote the total number of classified instances, then the model accuracy is given by: $\frac{T_P + T_N}{K}$.

This measure applies to binary and multi-class classifiers.

- The **confusion_matrix** result is a confusion matrix for a binary classifier model, formatted for human readability.

Notes

The **confusion_matrix** is not yet implemented for multi-class classifiers.

25.1.10 *EdgeFrame* column_median

column_median (*self*, *data_column*, *weights_column=None*)

Calculate the (weighted) median of a column.

Parameters **data_column** : unicode

The column whose median is to be calculated.

weights_column : unicode (default=None)

The column that provides weights (frequencies) for the median calculation. Must contain numerical data. Default is all items have a weight of 1.

Returns : dict

varies The median of the values. If a weight column is provided and no weights are finite numbers greater than 0, None is returned. The type of the median returned is the same as the contents of the data column, so a column of Longs will result in a Long median and a column of Floats will result in a Float median.

The median is the least value X in the range of the distribution so that the cumulative weight of values strictly below X is strictly less than half of the total weight and the cumulative weight of values up to and including X is greater than or equal to one-half of the total weight.

All data elements of weight less than or equal to 0 are excluded from the calculation, as are all data elements whose weight is NaN or infinite. If a weight column is provided and no weights are finite numbers greater than 0, None is returned.

25.1.11 *EdgeFrame* column_mode

column_mode (*self*, *data_column*, *weights_column=None*, *max_modes_returned=None*)

Evaluate the weights assigned to rows.

Parameters **data_column** : unicode

Name of the column supplying the data.

weights_column : unicode (default=None)

Name of the column supplying the weights. Default is all items have weight of 1.

max_modes_returned : int32 (default=None)

Maximum number of modes returned. Default is 1.

Returns : dict

dict Dictionary containing summary statistics. The data returned is composed of multiple components:

mode [A mode is a data element of maximum net weight.] A set of modes is returned. The empty set is returned when the sum of the weights is 0. If the number of modes is less than or equal to the parameter `max_modes_returned`, then all modes of the data are returned. If the number of modes is greater than the `max_modes_returned` parameter, only the first `max_modes_returned` many modes (per a canonical ordering) are returned.

weight_of_mode [Weight of a mode.] If there are no data elements of finite weight greater than 0, the weight of the mode is 0. If no weights column is given, this is the number of appearances of each mode.

total_weight [Sum of all weights in the weight column.] This is the row count if no weights are given. If no weights column is given, this is the number of rows in the table with non-zero weight.

mode_count [The number of distinct modes in the data.] In the case that the data is very multimodal, this number may exceed `max_modes_returned`.

Calculate the modes of a column. A mode is a data element of maximum weight. All data elements of weight less than or equal to 0 are excluded from the calculation, as are all data elements whose weight is NaN or infinite. If there are no data elements of finite weight greater than 0, no mode is returned.

Because data distributions often have multiple modes, it is possible for a set of modes to be returned. By default, only one is returned, but by setting the optional parameter `max_modes_returned`, a larger number of modes can be returned.

25.1.12 *EdgeFrame* `column_names`

column_names

Column identifications in the current frame.

Parameters

Returns : list

list of names of all the frame's columns

Given a Frame object, *my_frame* accessing a frame. To get the column names:

```
>>> my_columns = my_frame.column_names
>>> print my_columns
```

Now, given there are three columns *col1*, *col2*, and *col3*, the result is:

```
["col1", "col2", "col3"]
```

25.1.13 *EdgeFrame* `column_summary_statistics`

column_summary_statistics (*self*, *data_column*, *weights_column=None*,
use_population_variance=None)

Calculate multiple statistics for a column.

Parameters **data_column** : unicode

The column to be statistically summarized. Must contain numerical data; all NaNs and infinite values are excluded from the calculation.

weights_column : unicode (default=None)

Name of column holding weights of column values.

use_population_variance : bool (default=None)

If true, the variance is calculated as the population variance. If false, the variance calculated as the sample variance. Because this option affects the variance, it affects the standard deviation and the confidence intervals as well. Default is false.

Returns : dict

dict Dictionary containing summary statistics. The data returned is composed of multiple components:

mean [[double | None]] Arithmetic mean of the data.

geometric_mean [[double | None]] Geometric mean of the data. None when there is a data element ≤ 0 , 1.0 when there are no data elements.

variance [[double | None]] None when there are ≤ 1 many data elements. Sample variance is the weighted sum of the squared distance of each data element from the weighted mean, divided by the total weight minus 1. None when the sum of the weights is ≤ 1 . Population variance is the weighted sum of the squared distance of each data element from the weighted mean, divided by the total weight.

standard_deviation [[double | None]] The square root of the variance. None when sample variance is being used and the sum of weights is ≤ 1 .

total_weight [long] The count of all data elements that are finite numbers. (In other words, after excluding NaNs and infinite values.)

minimum [[double | None]] Minimum value in the data. None when there are no data elements.

maximum [[double | None]] Maximum value in the data. None when there are no data elements.

mean_confidence_lower [[double | None]] Lower limit of the 95% confidence interval about the mean. Assumes a Gaussian distribution. None when there are no elements of positive weight.

mean_confidence_upper [[double | None]] Upper limit of the 95% confidence interval about the mean. Assumes a Gaussian distribution. None when there are no elements of positive weight.

bad_row_count [[double | None]] The number of rows containing a NaN or infinite value in either the data or weights column.

good_row_count [[double | None]] The number of rows not containing a NaN or infinite value in either the data or weights column.

positive_weight_count [[double | None]] The number of valid data elements with weight > 0 . This is the number of entries used in the statistical calculation.

non_positive_weight_count [[double | None]] The number valid data elements with finite weight ≤ 0 .

Notes

Sample Variance Sample Variance is computed by the following formula:

$$\left(\frac{1}{W-1}\right) * \sum_i (x_i - M)^2$$

where W is sum of weights over valid elements of positive weight, and M is the weighted mean.

Population Variance Population Variance is computed by the following formula:

$$\left(\frac{1}{W}\right) * \sum_i (x_i - M)^2$$

where W is sum of weights over valid elements of positive weight, and M is the weighted mean.

Standard Deviation The square root of the variance.

Logging Invalid Data A row is bad when it contains a NaN or infinite value in either its data or weights column. In this case, it contributes to `bad_row_count`; otherwise it contributes to good row count.

A good row can be skipped because the value in its weight column is less than or equal to 0. In this case, it contributes to `non_positive_weight_count`, otherwise (when the weight is greater than 0) it contributes to `valid_data_weight_pair_count`.

Equations `bad_row_count + good_row_count = # rows in the frame`

`positive_weight_count + non_positive_weight_count = good_row_count`

In particular, when no weights column is provided and all weights are 1.0,

`non_positive_weight_count = 0` and `positive_weight_count = good_row_count`

25.1.14 *EdgeFrame* `compute_misplaced_score`

`compute_misplaced_score` (*self*, *gravity*)

Parameters `gravity` : float64

Similarity measure for computing tension between 2 connected items

Returns : <bound method `AtkEntityType.__name__` of

<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

25.1.15 *EdgeFrame* `copy`

`copy` (*self*, *columns=None*, *where=None*, *name=None*)

Create new frame from current frame.

Parameters `columns` : str | list of str | dict (default=None)

If not None, the copy will only include the columns specified. If dict, the string pairs represent a column renaming, {`source_column_name`: `destination_column_name`}

where : function (default=None)

If not None, only those rows for which the UDF evaluates to True will be copied.

name : str (default=None)

Name of the copied frame

Returns : Frame

A new Frame of the copied data.

Copy frame or certain frame columns entirely or filtered. Useful for frame query.

Examples

Build a Frame from a csv file with 5 million rows of data; call the frame “cust”:

```
>>> my_frame = ta.Frame(source="my_data.csv")
>>> my_frame.name("cust")
```

Given the frame has columns *id*, *name*, *hair*, and *shoe*. Copy it to a new frame:

```
>>> your_frame = my_frame.copy()
```

Now we have two frames of data, each with 5 million rows. Checking the names:

```
>>> print my_frame.name()
>>> print your_frame.name()
```

Gives the results:

```
"cust"
"frame_75401b7435d7132f5470ba35..."
```

Now, let’s copy *some* of the columns from the original frame:

```
>>> our_frame = my_frame.copy(['id', 'hair'])
```

Our new frame now has two columns, *id* and *hair*, and has 5 million rows. Let’s try that again, but this time change the name of the *hair* column to *color*:

```
>>> last_frame = my_frame.copy(('id': 'id', 'hair': 'color'))
```

25.1.16 *EdgeFrame* correlation

correlation (*self*, *data_column_names*)

Calculate correlation for two columns of current frame.

Parameters *data_column_names* : list

The names of 2 columns from which to compute the correlation.

Returns : dict

Pearson correlation coefficient of the two columns.

This method applies only to columns containing numerical data.

25.1.17 *EdgeFrame* correlation_matrix

correlation_matrix (*self*, *data_column_names*, *matrix_name=None*)

Calculate correlation matrix for two or more columns.

Parameters *data_column_names* : list

The names of the columns from which to compute the matrix.

matrix_name : unicode (default=None)

The name for the returned matrix Frame.

Returns : <bound method *AtkEntityType.__name__* of
<trustedanalytics.rest.jsonschema.*AtkEntityType* object at 0x7f6675773090>>

A Frame with the matrix of the correlation values for the columns.

This method applies only to columns containing numerical data.

25.1.18 *EdgeFrame* count

count (*self*, *where*)

Counts the number of rows which meet given criteria.

Parameters *where* : function

UDF which evaluates a row to a boolean

Returns : int

number of rows for which the where UDF evaluated to True.

25.1.19 *EdgeFrame* covariance

covariance (*self*, *data_column_names*)

Calculate covariance for exactly two columns.

Parameters *data_column_names* : list

The names of two columns from which to compute the covariance.

Returns : dict

Covariance of the two columns.

This method applies only to columns containing numerical data.

25.1.20 *EdgeFrame* covariance_matrix

covariance_matrix (*self*, *data_column_names*, *matrix_name=None*)

Calculate covariance matrix for two or more columns.

Parameters *data_column_names* : list

The names of the column from which to compute the matrix. Names should refer to a single column of type vector, or two or more columns of numeric scalars.

matrix_name : unicode (default=None)

The name of the new matrix.

Returns : <bound method *AtkEntityType.__name__* of
<trustedanalytics.rest.jsonschema.*AtkEntityType* object at 0x7f6675773090>>

A matrix with the covariance values for the columns.

This function applies only to columns containing numerical data.

25.1.21 *EdgeFrame* cumulative_percent

cumulative_percent (*self*, *sample_col*)

[BETA] Add column to frame with cumulative percent sum.

Parameters *sample_col* : unicode

The name of the column from which to compute the cumulative percent sum.

Returns : *_Unit*

A cumulative percent sum is computed by sequentially stepping through the rows, observing the column values and keeping track of the current percentage of the total sum accounted for at the current value.

Notes

This method applies only to columns containing numerical data. Although this method will execute for columns containing negative values, the interpretation of the result will change (for example, negative percentages).

25.1.22 *EdgeFrame* cumulative_sum

cumulative_sum (*self*, *sample_col*)

[BETA] Add column to frame with cumulative percent sum.

Parameters *sample_col* : unicode

The name of the column from which to compute the cumulative sum.

Returns : *_Unit*

A cumulative sum is computed by sequentially stepping through the rows, observing the column values and keeping track of the cumulative sum for each value.

Notes

This method applies only to columns containing numerical data.

25.1.23 *EdgeFrame* dot_product

dot_product (*self*, *left_column_names*, *right_column_names*, *dot_product_column_name*, *default_left_values=None*, *default_right_values=None*)
[ALPHA] Calculate dot product for each row in current frame.

Parameters *left_column_names* : list

Names of columns used to create the left vector (A) for each row. Names should refer to a single column of type vector, or two or more columns of numeric scalars.

right_column_names : list

Names of columns used to create right vector (B) for each row. Names should refer to a single column of type vector, or two or more columns of numeric scalars.

dot_product_column_name : unicode

Name of column used to store the dot product.

default_left_values : list (default=None)

Default values used to substitute null values in left vector. Default is None.

default_right_values : list (default=None)

Default values used to substitute null values in right vector. Default is None.

Returns : _Unit

Calculate the dot product for each row in a frame using values from two equal-length sequences of columns.

Dot product is computed by the following formula:

The dot product of two vectors $A = [a_1, a_2, \dots, a_n]$ and $B = [b_1, b_2, \dots, b_n]$ is $a_1 * b_1 + a_2 * b_2 + \dots + a_n * b_n$. The dot product for each row is stored in a new column in the existing frame.

Notes

If *default_left_values* or *default_right_values* are not specified, any null values will be replaced by zeros.

25.1.24 *EdgeFrame* download

download (*self*, *n=100*, *offset=0*, *columns=None*)
Download a frame from the server into client workspace.

Parameters *n* : int (default=100)

The number of rows to download to the client

offset : int (default=0)

The number of rows to skip before copying

columns : list (default=None)

Column filter, the names of columns to be included (default is all columns)

Returns : pandas.DataFrame

A new pandas dataframe object containing the downloaded frame data

Copies an trustedanalytics Frame into a Pandas DataFrame.

Examples

Frame *my_frame* accesses a frame with millions of rows of data. Get a sample of 500 rows:

```
>>> pandas_frame = my_frame.download( 500 )
```

We now have a new frame accessed by a pandas DataFrame *pandas_frame* with a copy of the first 500 rows of the original frame.

If we use the method with an offset like:

```
>>> pandas_frame = my_frame.take( 500, 100 )
```

We end up with a new frame accessed by the pandas DataFrame *pandas_frame* again, but this time it has a copy of rows 101 to 600 of the original frame.

25.1.25 EdgeFrame drop_columns

drop_columns (*self*, *columns*)

Remove columns from the frame.

Parameters **columns** : list

Column name OR list of column names to be removed from the frame.

Returns : _Unit

The data from the columns is lost.

Notes

It is not possible to delete all columns from a frame. At least one column needs to remain. If it is necessary to delete all columns, then delete the frame.

25.1.26 EdgeFrame drop_duplicates

drop_duplicates (*self*, *unique_columns=None*)

Modify the current frame, removing duplicate rows.

Parameters **unique_columns** : None (default=None)

Returns : _Unit

Remove data rows which are the same as other rows. The entire row can be checked for duplication, or the search for duplicates can be limited to one or more columns. This modifies the current frame.

25.1.27 *EdgeFrame* drop_rows

drop_rows (*self*, *predicate*)

Erase any row in the current frame which qualifies.

Parameters **predicate** : function

UDF which evaluates a row to a boolean; rows that answer True are dropped from the Frame

Examples

For this example, `my_frame` is a `Frame` object accessing a frame with lots of data for the attributes of `lions`, `tigers`, and `ligers`. Get rid of the `lions` and `tigers`:

```
>>> my_frame.drop_rows(lambda row: row.animal_type == "lion" or
...                     row.animal_type == "tiger")
```

Now the frame only has information about `ligers`.

More information on a UDF can be found at [Python User Functions](#).

25.1.28 *EdgeFrame* ecdf

ecdf (*self*, *column*, *result_frame_name=None*)

Builds new frame with columns for data and distribution.

Parameters **column** : unicode

The name of the input column containing sample.

result_frame_name : unicode (default=None)

A name for the resulting frame which is created by this operation.

Returns : <bound method `AtkEntityType.__name__` of

<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A new `Frame` containing each distinct value in the sample and its corresponding ECDF value.

Generates the *empirical cumulative distribution* for the input column.

25.1.29 *EdgeFrame* entropy

entropy (*self*, *data_column*, *weights_column=None*)

Calculate the Shannon entropy of a column.

Parameters **data_column** : unicode

The column whose entropy is to be calculated.

weights_column : unicode (default=None)

The column that provides weights (frequencies) for the entropy calculation. Must contain numerical data. Default is using uniform weights of 1 for all items.

Returns : dict

Entropy.

The data column is weighted via the weights column. All data elements of weight ≤ 0 are excluded from the calculation, as are all data elements whose weight is NaN or infinite. If there are no data elements with a finite weight greater than 0, the entropy is zero.

25.1.30 *EdgeFrame* export_to_csv

export_to_csv (*self*, *folder_name*, *separator=None*, *count=None*, *offset=None*)

Write current frame to HDFS in csv format.

Parameters **folder_name** : unicode

The HDFS folder path where the files will be created.

separator : None (default=None)

count : int32 (default=None)

The number of records you want. Default, or a non-positive value, is the whole frame.

offset : int32 (default=None)

The number of rows to skip before exporting to the file. Default is zero (0).

Returns : _Unit

Export the frame to a file in csv format as a Hadoop file.

25.1.31 *EdgeFrame* export_to_hbase

export_to_hbase (*self*, *table_name*, *key_column_name=None*, *family_name=None*)

Write current frame to HBase table.

Parameters **table_name** : unicode

The name of the HBase table that will contain the exported frame

key_column_name : unicode (default=None)

The name of the column to be used as row key in hbase table

family_name : unicode (default=None)

The family name of the HBase table that will contain the exported frame

Returns : _Unit

Table must exist in HBase. Export of Vectors is not currently supported.

25.1.32 *EdgeFrame* export_to_hive

export_to_hive (*self*, *table_name*)

Write current frame to Hive table.

Parameters *table_name* : unicode

The name of the Hive table that will contain the exported frame

Returns : `_Unit`

Table must not exist in Hive. Export of Vectors is not currently supported.

25.1.33 *EdgeFrame* export_to_jdbc

export_to_jdbc (*self*, *table_name*, *connector_type=None*, *url=None*, *driver_name=None*, *query=None*)

Write current frame to Jdbc table.

Parameters *table_name* : unicode

jdbc table name

connector_type : unicode (default=None)

(optional) jdbc connector type

url : unicode (default=None)

(optional) connection url (includes server name, database name, user acct and password)

driver_name : unicode (default=None)

(optional) driver name

query : unicode (default=None)

(optional) query for filtering. Not supported yet.

Returns : `_Unit`

Table will be created or appended to. Export of Vectors is not currently supported.

25.1.34 *EdgeFrame* export_to_json

export_to_json (*self*, *folder_name*, *count=None*, *offset=None*)

Write current frame to HDFS in JSON format.

Parameters *folder_name* : unicode

The HDFS folder path where the files will be created.

count : int32 (default=None)

The number of records you want. Default, or a non-positive value, is the whole frame.

offset : int32 (default=None)

The number of rows to skip before exporting to the file. Default is zero (0).

Returns : `_Unit`

Export the frame to a file in JSON format as a Hadoop file.

25.1.35 *EdgeFrame* filter

filter (*self*, *predicate*)

Select all rows which satisfy a predicate.

Parameters **predicate** : function

UDF which evaluates a row to a boolean; rows that answer False are dropped from the Frame

Modifies the current frame to save defined rows and delete everything else.

Examples

For this example, *my_frame* is a Frame object with lots of data for the attributes of lizards, frogs, and snakes. Get rid of everything, except information about lizards and frogs:

```
>>> def my_filter(row):  
...     return row['animal_type'] == 'lizard' or  
...     row['animal_type'] == "frog"  
  
>>> my_frame.filter(my_filter)
```

The frame now only has data about lizards and frogs.

More information on a UDF can be found at [Python User Functions](#).

25.1.36 *EdgeFrame* flatten_column

flatten_column (*self*, *column*, *delimiter=None*)

Spread data to multiple rows based on cell data.

Parameters **column** : unicode

The column to be flattened.

delimiter : unicode (default=None)

The delimiter string. Default is comma (,).

Returns : `_Unit`

Splits cells in the specified column into multiple rows according to a string delimiter. New rows are a full copy of the original row, but the specified column only contains one value. The original row is deleted.

25.1.37 *EdgeFrame* `get_error_frame`

`get_error_frame` (*self*)

Get a frame with error recordings.

Parameters

When a frame is created, another frame is transparently created to capture parse errors.

Returns **Frame** : error frame object

A new object accessing a frame that contains the parse errors of the currently active Frame or None if no error frame exists.

25.1.38 *EdgeFrame* `group_by`

`group_by` (*self*, *group_by_columns*, *aggregation_arguments=None*)

[BETA] Create summarized frame.

Parameters **`group_by_columns`** : list

Column name or list of column names

`aggregation_arguments` : dict (default=None)

Aggregation function based on entire row, and/or dictionaries (one or more) of { column name str : aggregation function(s) }.

Returns : Frame

A new frame with the results of the `group_by`

Creates a new frame and returns a Frame object to access it. Takes a column or group of columns, finds the unique combination of values, and creates unique rows with these column values. The other columns are combined according to the aggregation argument(s).

Notes

- Column order is not guaranteed when columns are added
- The column names created by aggregation functions in the new frame are the original column name appended with the ‘_’ character and the aggregation function. For example, if the original field is *a* and the function is *avg*, the resultant column is named *a_avg*.
- An aggregation argument of *count* results in a column named *count*.
- The aggregation function *agg.count* is the only full row aggregation function supported at this time.
- Aggregation currently supports using the following functions:
 - avg
 - count
 - count_distinct
 - max
 - min

- stdev
- sum
- var (see glossary *Bias vs Variance*)

Examples

For setup, we will use a Frame *my_frame* accessing a frame with a column *a*:

```
>>> my_frame.inspect()

a:str
/-----/
cat
apple
bat
cat
bat
cat
```

Create a new frame, combining similar values of column *a*, and count how many of each value is in the original frame:

```
>>> new_frame = my_frame.group_by('a', agg.count)
>>> new_frame.inspect()

a:str      count:int
/-----/
cat         3
apple       1
bat         2
```

In this example, ‘*my_frame*’ is accessing a frame with three columns, *a*, *b*, and *c*:

```
>>> my_frame.inspect()

a:int  b:str  c:float
/-----/
1      alpha  3.0
1      bravo  5.0
1      alpha  5.0
2      bravo  8.0
2      bravo 12.0
```

Create a new frame from this data, grouping the rows by unique combinations of column *a* and *b*. Average the value in *c* for each group:

```
>>> new_frame = my_frame.group_by(['a', 'b'], {'c' : agg.avg})
>>> new_frame.inspect()

a:int  b:str  c_avg:float
/-----/
1      alpha  4.0
1      bravo  5.0
2      bravo 10.0
```

For this example, we use *my_frame* with columns *a*, *c*, *d*, and *e*:

```
>>> my_frame.inspect()

  a:str   c:int   d:float e:int
/-----/
ape      1      4.0     9
ape      1      8.0     8
big      1      5.0     7
big      1      6.0     6
big      1      8.0     5
```

Create a new frame from this data, grouping the rows by unique combinations of column *a* and *c*. Count each group; for column *d* calculate the average, sum and minimum value. For column *e*, save the maximum value:

```
>>> new_frame = my_frame.group_by(['a', 'c'], agg.count,
... {'d': [agg.avg, agg.sum, agg.min], 'e': agg.max})

  a   c   count  d_avg  d_sum  d_min  e_max
str int  int    float  float  float  int
/-----/
ape  1   2     6.0    12.0   4.0    9
big  1   3     6.333  19.0   5.0    7
```

For further examples, see [Group by \(and aggregate\):](#).

25.1.39 EdgeFrame histogram

histogram (*self*, *column_name*, *num_bins*=None, *weight_column_name*=None, *bin_type*='equalwidth')
[BETA] Compute the histogram for a column in a frame.

Parameters **column_name** : unicode

Name of column to be evaluated.

num_bins : int32 (default=None)

Number of bins in histogram. Default is Square-root choice will be used (in other words $\text{math.floor}(\text{math.sqrt}(\text{frame.row_count}))$).

weight_column_name : unicode (default=None)

Name of column containing weights. Default is all observations are weighted equally.

bin_type : unicode (default=equalwidth)

The type of binning algorithm to use: ["equalwidth"|"equaldepth"] Defaults is "equalwidth".

Returns : dict

histogram A Histogram object containing the result set. The data returned is composed of multiple components:

cutoffs [array of float] A list containing the edges of each bin.

hist [array of float] A list containing count of the weighted observations found in each bin.

density [array of float] A list containing a decimal containing the percentage of observations found in the total set per bin.

Compute the histogram of the data in a column. The returned value is a Histogram object containing 3 lists one each for: the cutoff points of the bins, size of each bin, and density of each bin.

Notes

The `num_bins` parameter is considered to be the maximum permissible number of bins because the data may dictate fewer bins. With equal depth binning, for example, if the column to be binned has 10 elements with only 2 distinct values and the `num_bins` parameter is greater than 2, then the number of actual number of bins will only be 2. This is due to a restriction that elements with an identical value must belong to the same bin.

25.1.40 *EdgeFrame* inspect

inspect (*self*, *n=10*, *offset=0*, *columns=None*, *wrap=None*, *truncate=None*, *round=None*, *width=80*, *margin=None*)

Prints the frame data in readable format.

Parameters **n** : int (default=10)

The number of rows to print.

offset : int (default=0)

The number of rows to skip before printing.

columns : int (default=None)

Filter columns to be included. By default, all columns are included

wrap : int or 'stripes' (default=None)

If set to 'stripes' then inspect prints rows in stripes; if set to an integer N, rows will be printed in clumps of N columns, where the columns are wrapped

truncate : int (default=None)

If set to integer N, all strings will be truncated to length N, including a tagged ellipses

round : int (default=None)

If set to integer N, all floating point numbers will be rounded and truncated to N digits

width : int (default=80)

If set to integer N, the print out will try to honor a max line width of N

margin : int (default=None)

('stripes' mode only) If set to integer N, the margin for printing names in a stripe will be limited to N characters

Examples

Given a frame of data and a Frame to access it. To look at the first 4 rows of data:

```
>>> print my_frame.inspect(4)

column defs ->  animal:str  name:str  age:int  weight:float
               /-----/
frame data ->  human      George    8        542.5
               human      Ursula     6        495.0
```


ape	Ape	41	400.0
elephant	Shep	5	8630.0

For other examples, see *Inspect The Data*.

25.1.41 *EdgeFrame* join

join (*self*, *right*, *left_on*, *right_on*=None, *how*=‘inner’, *name*=None)
 [BETA] Join operation on one or two frames, creating a new frame.

Parameters **right** : Frame

Another frame to join with

left_on : str

Name of the column in the left frame used to match up the two frames.

right_on : str (default=None)

Name of the column in the right frame used to match up the two frames. Default is the same as the left frame.

how : str (default=inner)

How to qualify the data to be joined together. Must be one of the following: ‘left’, ‘right’, ‘inner’, ‘outer’. Default is ‘inner’

name : str (default=None)

Name of the result grouped frame

Returns : Frame

A new frame with the results of the join

Create a new frame from a SQL JOIN operation with another frame. The frame on the ‘left’ is the currently active frame. The frame on the ‘right’ is another frame. This method takes a column in the left frame and matches its values with a column in the right frame. Using the default ‘how’ option [‘inner’] will only allow data in the resultant frame if both the left and right frames have the same value in the matching column. Using the ‘left’ ‘how’ option will allow any data in the resultant frame if it exists in the left frame, but will allow any data from the right frame if it has a value in its column which matches the value in the left frame column. Using the ‘right’ option works similarly, except it keeps all the data from the right frame and only the data from the left frame when it matches. The ‘outer’ option provides a frame with data from both frames where the left and right frames did not have the same value in the matching column.

Notes

When a column is named the same in both frames, it will result in two columns in the new frame. The column from the *left* frame (originally the current frame) will be copied and the column name will have the string “_L” added to it. The same thing will happen with the column from the *right* frame, except its name has the string “_R” appended. The order of columns after this method is called is not guaranteed.

It is recommended that you rename the columns to meaningful terms prior to using the `join` method. Keep in mind that unicode in column names will likely cause the `drop_frames()` method (and others) to fail!

Examples

For this example, we will use a Frame *my_frame* accessing a frame with columns *a*, *b*, *c*, and a Frame *your_frame* accessing a frame with columns *a*, *d*, *e*. Join the two frames keeping only those rows having the same value in column *a*:

```
>>> print my_frame.inspect()

  a:unicode   b:unicode   c:unicode
/-----/
alligator    bear        cat
apple        berry       cantaloupe
auto         bus         car
mirror       frog        ball

>>> print your_frame.inspect()

  b:unicode   c:int    d:unicode
/-----/
berry        5218    frog
blue         0      log
bus          871    dog

>>> joined_frame = my_frame.join(your_frame, 'b', how='inner')
```

Now, *joined_frame* is a Frame accessing a frame with the columns *a*, *b*, *c_L*, *c_R*, and *d*. The data in the new frame will be from the rows where column 'a' was the same in both frames.

```
>>> print joined_frame.inspect()

  a:unicode   b:unicode   c_L:unicode   c_R:int64   d:unicode
/-----/
apple        berry       cantaloupe    5218        frog
auto         bus         car           871         dog
```

More examples can be found in the [user manual](#).

25.1.42 EdgeFrame loadhbase

loadhbase (*self*, *table_name*, *schema*, *start_tag*=None, *end_tag*=None)

Append data from an hBase table into an existing (possibly empty) FrameRDD

Parameters *table_name* : unicode

hbase table name

schema : list

hbase schema as a list of tuples (columnFamily, columnName, dataType for cell value)

start_tag : unicode (default=None)

optional start tag for filtering

end_tag : unicode (default=None)

optional end tag for filtering

Returns : <bound method `AtkEntityType.__name__` of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
 the initial `FrameRDD` with the `hbase` data appended

Append data from an `hBase` table into an existing (possibly empty) `FrameRDD`

25.1.43 *EdgeFrame* loadhive

loadhive (*self*, *query*)

Append data from a `hive` table into an existing (possibly empty) frame

Parameters **query** : unicode

 Initial query to run at load time

Returns : <bound method `AtkEntityType.__name__` of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
 the initial frame with the `hive` data appended

Append data from a `hive` table into an existing (possibly empty) frame

25.1.44 *EdgeFrame* loadjdbc

loadjdbc (*self*, *table_name*, *connector_type=None*, *url=None*, *driver_name=None*, *query=None*)

Append data from a `Jdbc` table into an existing (possibly empty) frame

Parameters **table_name** : unicode

 table name

connector_type : unicode (default=None)

 (optional) connector type

url : unicode (default=None)

 (optional) connection url (includes server name, database name, user acct and password)

driver_name : unicode (default=None)

 (optional) driver name

query : unicode (default=None)

 (optional) query for filtering. Not supported yet.

Returns : <bound method `AtkEntityType.__name__` of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
 the initial frame with the `Jdbc` data appended

Append data from a `Jdbc` table into an existing (possibly empty) frame

25.1.45 *EdgeFrame* name

name

Set or get the name of the frame object.

Parameters

Change or retrieve frame object identification. Identification names must start with a letter and are limited to alphanumeric characters and the `_` character.

Examples

```
>>> my_frame.name
"csv_data"

>>> my_frame.name = "cleaned_data"
>>> my_frame.name
"cleaned_data"
```

25.1.46 *EdgeFrame* quantiles

quantiles (*self*, *column_name*, *quantiles*)

New frame with Quantiles and their values.

Parameters **column_name** : unicode

The column to calculate quantiles.

quantiles : list

What is being requested.

Returns : <bound method `AtkEntityType.__name__` of

<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A new frame with two columns (float64): requested Quantiles and their respective values.

Calculate quantiles on the given column.

25.1.47 *EdgeFrame* rename_columns

rename_columns (*self*, *names*)

Rename columns for edge frame.

Parameters **names** : None**Returns** : `_Unit`

25.1.48 *EdgeFrame* row_count

row_count

Number of rows in the current frame.

Parameters

Returns : int

The number of rows in the frame

Get the number of rows:

```
>>> my_frame.row_count
```

The result given is:

```
81734
```

25.1.49 *EdgeFrame* schema

schema

Current frame column names and types.

Parameters

Returns : list

list of tuples of the form (<column name>, <data type>)

The schema of the current frame is a list of column names and associated data types. It is retrieved as a list of tuples. Each tuple has the name and data type of one of the frame's columns.

Examples

Given that we have an existing data frame *my_data*, create a Frame, then show the frame schema:

```
>>> BF = ta.get_frame('my_data')
>>> print BF.schema
```

The result is:

```
[("col1", str), ("col2", numpy.int32)]
```

25.1.50 *EdgeFrame* sort

sort (*self*, *columns*, *ascending=True*)

[BETA] Sort the data in a frame.

Parameters **columns** : str | list of str | list of tuples

Either a column name, a list of column names, or a list of tuples where each tuple is a name and an ascending bool value.

ascending : bool (default=True)

True for ascending, False for descending.

Sort a frame by column values either ascending or descending.

Examples

Sort a single column:

```
>>> frame.sort('column_name')
```

Sort a single column ascending:

```
>>> frame.sort('column_name', True)
```

Sort a single column descending:

```
>>> frame.sort('column_name', False)
```

Sort multiple columns:

```
>>> frame.sort(['col1', 'col2'])
```

Sort multiple columns ascending:

```
>>> frame.sort(['col1', 'col2'], True)
```

Sort multiple columns descending:

```
>>> frame.sort(['col1', 'col2'], False)
```

Sort multiple columns: 'col1' ascending and 'col2' descending:

```
>>> frame.sort([ ('col1', True), ('col2', False) ])
```

25.1.51 *EdgeFrame* sorted_k

sorted_k (*self*, *k*, *column_names_and_ascending*, *reduce_tree_depth=None*)

[ALPHA] Get a sorted subset of the data.

Parameters **k** : int32

Number of sorted records to return.

column_names_and_ascending : list

Column names to sort by, and true to sort column by ascending order, or false for descending order.

reduce_tree_depth : int32 (default=None)

Advanced tuning parameter which determines the depth of the reduce-tree for the sorted_k plugin. This plugin uses Spark's treeReduce() for scalability. The default depth is 2.

Returns : <bound method *AtkEntityType.__name__* of

<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A new frame with the first k sorted rows from the original frame.

Take the first k (sorted) rows for the currently active Frame. Rows are sorted by column values in either ascending or descending order.

Returning the first k (sorted) rows is more efficient than sorting the entire frame when k is much smaller than the number of rows in the frame.

Notes

The number of sorted rows (k) should be much smaller than the number of rows in the original frame.

In particular:

1. The number of sorted rows (k) returned should fit in Spark driver memory.

The maximum size of serialized results that can fit in the Spark driver is set by the Spark configuration parameter `spark.driver.maxResultSize`.

2. If you encounter a Kryo buffer overflow exception, increase the Spark

configuration parameter `spark.kryoserializer.buffer.max.mb`.

3. Use `Frame.sort()` instead if the number of sorted rows (k) is

very large (i.e., cannot fit in Spark driver memory).

25.1.52 *EdgeFrame* status

status

Current frame life cycle status.

Parameters

Returns : str

Status of the frame

One of three statuses: Active, Deleted, Deleted_Final Active: Frame is available for use Deleted: Frame has been scheduled for deletion can be unscheduled by modifying Deleted_Final: Frame's backend files have been removed from disk.

Examples

Given that we have an existing data frame `my_data`, create a Frame, then show the frame schema:

```
>>> BF = ta.get_frame('my_data')
>>> print BF.status
```

The result is:

```
u'Active'
```

25.1.53 *EdgeFrame* take

take (*self*, *n*, *offset=0*, *columns=None*)

Get data subset.

Parameters *n* : int

The number of rows to copy to the client from the frame.

offset : int (default=0)

The number of rows to skip before starting to copy

columns : str | iterable of str (default=None)

If not None, only the given columns' data will be provided. By default, all columns are included

Returns : list

A list of lists, where each contained list is the data for one row.

Take a subset of the currently active Frame.

Notes

The data is considered 'unstructured', therefore taking a certain number of rows, the rows obtained may be different every time the command is executed, even if the parameters do not change.

Examples

Frame *my_frame* accesses a frame with millions of rows of data. Get a sample of 5000 rows:

```
>>> my_data_list = my_frame.take( 5000 )
```

We now have a list of data from the original frame.

```
>>> print my_data_list

[[ 1, "text", 3.1415962 ]
 [ 2, "bob", 25.0 ]
 [ 3, "weave", .001 ]
 ...]
```

If we use the method with an offset like:

```
>>> my_data_list = my_frame.take( 5000, 1000 )
```

We end up with a new list, but this time it has a copy of the data from rows 1001 to 5000 of the original frame.

25.1.54 *EdgeFrame* tally

tally (*self*, *sample_col*, *count_val*)

[BETA] Count number of times a value is seen.

Parameters `sample_col` : unicode

The name of the column from which to compute the cumulative count.

`count_val` : unicode

The column value to be used for the counts.

Returns : `_Unit`

A cumulative count is computed by sequentially stepping through the rows, observing the column values and keeping track of the the number of times the specified *count_value* has been seen.

25.1.55 *EdgeFrame* `tally_percent`

tally_percent (*self*, *sample_col*, *count_val*)

[BETA] Compute a cumulative percent count.

Parameters `sample_col` : unicode

The name of the column from which to compute the cumulative sum.

`count_val` : unicode

The column value to be used for the counts.

Returns : `_Unit`

A cumulative percent count is computed by sequentially stepping through the rows, observing the column values and keeping track of the percentage of the total number of times the specified *count_value* has been seen up to the current value.

25.1.56 *EdgeFrame* `top_k`

top_k (*self*, *column_name*, *k*, *weights_column=None*)

Most or least frequent column values.

Parameters `column_name` : unicode

The column whose top (or bottom) K distinct values are to be calculated.

`k` : int32

Number of entries to return (If k is negative, return bottom k).

weights_column : unicode (default=None)

The column that provides weights (frequencies) for the topK calculation. Must contain numerical data. Default is 1 for all items.

Returns : <bound method `AtkEntityType.__name__` of

<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

An object with access to the frame of data.

Calculate the top (or bottom) K distinct values by count of a column. The column can be weighted. All data elements of weight ≤ 0 are excluded from the calculation, as are all data elements whose weight is NaN or infinite. If there are no data elements of finite weight > 0 , then topK is empty.

25.1.57 *EdgeFrame* `unflatten_column`

unflatten_column (*self*, *composite_key_column_names*, *delimiter=None*)

Compacts data from multiple rows based on cell data.

Parameters *composite_key_column_names* : list

name of the user column to be used as keys for unflattening.

delimiter : unicode (default=None)

separator for the data in the result columns. Default is comma (,).

Returns : `_Unit`

Groups together cells in all columns (less the composite key) using “,” as string delimiter. The original rows are deleted. The grouping takes place based on a composite key passed as arguments.

class EdgeFrame

A list of Edges owned by a Graph.

An `EdgeFrame` is similar to a `Frame` but with a few important differences:

- `EdgeFrames` are not instantiated directly by the user, instead they are created by defining an edge type in a graph
- Each row of an `EdgeFrame` represents an edge in a graph
- `EdgeFrames` have many of the same methods as `Frames` but not all
- `EdgeFrames` have extra methods not found on `Frames` (e.g. `add_edges()`)
- `EdgeFrames` have a dependency on one or two `VertexFrames` (adding an edge to an `EdgeFrame` requires either vertices to be present or for the user to specify `create_missing_vertices=True`)
- `EdgeFrames` have special system columns (`_eid`, `_label`, `_src_vid`, `_dest_vid`) that are maintained automatically by the system and cannot be modified by the user
- “Columns” on an `EdgeFrame` can also be thought of as “properties” on Edges

Attributes

<code>column_names</code>	Column identifications in the current frame.
<code>name</code>	Set or get the name of the frame object.
<code>row_count</code>	Number of rows in the current frame.
<code>schema</code>	Current frame column names and types.
<code>status</code>	Current frame life cycle status.

Methods

<code>__init__(self[, graph, label, src_vertex_label, ...])</code>	Examples
<code>add_columns(self, func, schema[, columns_accessed])</code>	Add columns to current frame.
<code>add_edges(self, source_frame, column_name_for_source_vertex_id, ...[, ...])</code>	Add edges to a graph.
<code>assign_sample(self, sample_percentages[, sample_labels, ...])</code>	Randomly group rows into user-defined classes.
<code>bin_column(self, column_name, cutoffs[, include_lowest, strict_binning, ...])</code>	Classify data into user-defined groups.

Table 25.1 – continued from previous page

<code>bin_column_equal_depth(self, column_name[, num_bins, ...])</code>	Classify column into groups with the same frequency.
<code>bin_column_equal_width(self, column_name[, num_bins, ...])</code>	Classify column into same-width groups.
<code>categorical_summary(self, *column_inputs)</code>	[ALPHA] Compute a summary of the data in a column.
<code>classification_metrics(self, label_column, pred_column[, ...])</code>	Model statistics of accuracy, precision, and others.
<code>column_median(self, data_column[, weights_column])</code>	Calculate the (weighted) median of a column.
<code>column_mode(self, data_column[, weights_column, max_modes_returned])</code>	Evaluate the weights assigned to rows.
<code>column_summary_statistics(self, data_column[, ...])</code>	Calculate multiple statistics for a column.
<code>compute_misplaced_score(self, gravity)</code>	
<code>copy(self[, columns, where, name])</code>	Create new frame from current frame.
<code>correlation(self, data_column_names)</code>	Calculate correlation for two columns of current frame.
<code>correlation_matrix(self, data_column_names[, matrix_name])</code>	Calculate correlation matrix for two or more columns.
<code>count(self, where)</code>	Counts the number of rows which meet given criteria.
<code>covariance(self, data_column_names)</code>	Calculate covariance for exactly two columns.
<code>covariance_matrix(self, data_column_names[, matrix_name])</code>	Calculate covariance matrix for two or more columns.
<code>cumulative_percent(self, sample_col)</code>	[BETA] Add column to frame with cumulative percent.
<code>cumulative_sum(self, sample_col)</code>	[BETA] Add column to frame with cumulative percent.
<code>dot_product(self, left_column_names, right_column_names, ...[, ...])</code>	[ALPHA] Calculate dot product for each row in current frame.
<code>download(self[, n, offset, columns])</code>	Download a frame from the server into client workspace.
<code>drop_columns(self, columns)</code>	Remove columns from the frame.
<code>drop_duplicates(self[, unique_columns])</code>	Modify the current frame, removing duplicate rows.
<code>drop_rows(self, predicate)</code>	Erase any row in the current frame which qualifies.
<code>ecdf(self, column[, result_frame_name])</code>	Builds new frame with columns for data and distribution.
<code>entropy(self, data_column[, weights_column])</code>	Calculate the Shannon entropy of a column.
<code>export_to_csv(self, folder_name[, separator, count, offset])</code>	Write current frame to HDFS in csv format.
<code>export_to_hbase(self, table_name[, key_column_name, family_name])</code>	Write current frame to HBase table.
<code>export_to_hive(self, table_name)</code>	Write current frame to Hive table.
<code>export_to_jdbc(self, table_name[, connector_type, url, driver_name, ...])</code>	Write current frame to Jdbc table.
<code>export_to_json(self, folder_name[, count, offset])</code>	Write current frame to HDFS in JSON format.
<code>filter(self, predicate)</code>	Select all rows which satisfy a predicate.
<code>flatten_column(self, column[, delimiter])</code>	Spread data to multiple rows based on cell data.
<code>get_error_frame(self)</code>	Get a frame with error recordings.
<code>group_by(self, group_by_columns, *aggregation_arguments)</code>	[BETA] Create summarized frame.
<code>histogram(self, column_name[, num_bins, weight_column_name, bin_type])</code>	[BETA] Compute the histogram for a column in a frame.
<code>inspect(self[, n, offset, columns, wrap, truncate, round, width, margin])</code>	Prints the frame data in readable format.
<code>join(self, right, left_on[, right_on, how, name])</code>	[BETA] Join operation on one or two frames, creating new frame.
<code>loadhbase(self, table_name, schema[, start_tag, end_tag])</code>	Append data from an hBase table into an existing (possibly empty) frame.
<code>loadhive(self, query)</code>	Append data from a hive table into an existing (possibly empty) frame.
<code>loadjdbc(self, table_name[, connector_type, url, driver_name, query])</code>	Append data from a Jdbc table into an existing (possibly empty) frame.
<code>quantiles(self, column_name, quantiles)</code>	New frame with Quantiles and their values.
<code>rename_columns(self, names)</code>	Rename columns for edge frame.
<code>sort(self, columns[, ascending])</code>	[BETA] Sort the data in a frame.
<code>sorted_k(self, k, column_names_and_ascending[, reduce_tree_depth])</code>	[ALPHA] Get a sorted subset of the data.
<code>take(self, n[, offset, columns])</code>	Get data subset.
<code>tally(self, sample_col, count_val)</code>	[BETA] Count number of times a value is seen.
<code>tally_percent(self, sample_col, count_val)</code>	[BETA] Compute a cumulative percent count.
<code>top_k(self, column_name, k[, weights_column])</code>	Most or least frequent column values.
<code>unflatten_column(self, composite_key_column_names[, delimiter])</code>	Compacts data from multiple rows based on cell data.

`__init__(self, graph=None, label=None, src_vertex_label=None, dest_vertex_label=None, directed=None)`

Examples

Parameters **graph** : ? (default=None)

graph these edges belong to

label : ? (default=None)

edge label

src_vertex_label : ? (default=None)

label of the source vertex type

dest_vertex_label : ? (default=None)

label of the destination vertex type

directed : ? (default=None)

directed or undirected

Returns : VertexFrame object

An object with access to the frame.

Given a data file `/movie.csv`, create a frame to match this data and move the data to the frame. Create an empty graph and define some vertex and edge types.

```
>>> my_csv = ta.CsvFile("/movie.csv", schema= [('user_id', int32),
...                                           ('user_name', str),
...                                           ('movie_id', int32),
...                                           ('movie_title', str),
...                                           ('rating', str)])

>>> my_frame = ta.Frame(my_csv)
>>> my_graph = ta.Graph()
>>> my_graph.define_vertex_type('users')
>>> my_graph.define_vertex_type('movies')
>>> my_graph.define_edge_type('ratings', 'users', 'movies', directed=True)
```

Add data to the graph from the frame:

```
>>> my_graph.vertices['users'].add_vertices(my_frame, 'user_id',
... [ 'user_name'])
>>> my_graph.vertices['movies'].add_vertices(my_frame, 'movie_id', ['movie_title'])
```

Create an edge frame from the graph, and add edge data from the frame.

```
>>> my_edge_frame = graph.edges['ratings']
>>> my_edge_frame.add_edges(my_frame, 'user_id', 'movie_id', ['rating'])
```

Retrieve a previously defined graph and retrieve an EdgeFrame from it:

```
>>> my_old_graph = ta.get_graph("your_graph")
>>> my_new_edge_frame = my_old_graph.edges["your_label"]
```

Calling methods on an EdgeFrame:

```
>>> my_new_edge_frame.inspect(20)
```

Copy an EdgeFrame to a frame using the copy method:

```
>>> my_new_frame = my_new_edge_frame.copy()
```

25.2 Frames VertexFrame

25.2.1 VertexFrame `__init__`

`__init__` (*self*, *source=None*, *graph=None*, *label=None*)

Examples

Parameters **source** : (default=None)

graph : (default=None)

label : (default=None)

Given a data file, create a frame, move the data to graph and then define a new VertexFrame and add data to it:

Retrieve a previously defined graph and retrieve a VertexFrame from it:

```
>>> my_graph = ta.get_graph("your_graph")
>>> my_vertex_frame = my_graph.vertices["your_label"]
```

Calling methods on a VertexFrame:

```
>>> my_vertex_frame.vertices["your_label"].inspect(20)
```

Convert a VertexFrame to a frame:

```
>>> new_Frame = my_vertex_frame.vertices["label"].copy()
```

25.2.2 VertexFrame `add_columns`

`add_columns` (*self*, *func*, *schema*, *columns_accessed=None*)

Add columns to current frame.

Parameters **func** : UDF

User-Defined Function (UDF) which takes the values in the row and produces a value, or collection of values, for the new cell(s).

schema : tuple | list of tuples

The schema for the results of the UDF, indicating the new column(s) to add. Each tuple provides the column name and data type, and is of the form (str, type).

columns_accessed : list (default=None)

List of columns which the UDF will access. This adds significant performance benefit if we know which column(s) will be needed to execute the UDF, especially when the frame has significantly more columns than those being used to evaluate the UDF.

Assigns data to column based on evaluating a function for each row.

Notes

- 1.The row UDF ('func') must return a value in the same format as specified by the schema. See [Python User Functions](#).
- 2.Unicode in column names is not supported and will likely cause the `drop_frames()` method (and others) to fail!

Examples

Given a Frame *my_frame* identifying a data frame with two int32 columns *column1* and *column2*. Add a third column *column3* as an int32 and fill it with the contents of *column1* and *column2* multiplied together:

```
>>> my_frame.add_columns(lambda row: row.column1*row.column2,  
... ('column3', int32))
```

The frame now has three columns, *column1*, *column2* and *column3*. The type of *column3* is an int32, and the value is the product of *column1* and *column2*.

Add a string column *column4* that is empty:

```
>>> my_frame.add_columns(lambda row: '', ('column4', str))
```

The Frame object *my_frame* now has four columns *column1*, *column2*, *column3*, and *column4*. The first three columns are int32 and the fourth column is str. Column *column4* has an empty string ('') in every row.

Multiple columns can be added at the same time. Add a column *a_times_b* and fill it with the contents of column *a* multiplied by the contents of column *b*. At the same time, add a column *a_plus_b* and fill it with the contents of column *a* plus the contents of column *b*:

```
>>> my_frame.add_columns(lambda row: [row.a * row.b, row.a +  
... row.b], [("a_times_b", float32), ("a_plus_b", float32)])
```

Two new columns are created, "a_times_b" and "a_plus_b", with the appropriate contents.

Given a frame of data and Frame *my_frame* points to it. In addition we have defined a UDF *func*. Run *func* on each row of the frame and put the result in a new int column *calculated_a*:

```
>>> my_frame.add_columns( func, ("calculated_a", int))
```

Now the frame has a column *calculated_a* which has been filled with the results of the UDF *func*.

A UDF must return a value in the same format as the column is defined. In most cases this is automatically the case, but sometimes it is less obvious. Given a UDF *function_b* which returns a value in a list, store the result in a new column *calculated_b*:

```
>>> my_frame.add_columns(function_b, ("calculated_b", float32))
```

This would result in an error because *function_b* is returning a value as a single element list like [2.4], but our column is defined as a tuple. The column must be defined as a list:

```
>>> my_frame.add_columns(function_b, [("calculated_b", float32)])
```

To run an optimized version of `add_columns`, `columns_accessed` parameter can be populated with the column names which are being accessed in UDF. This speeds up the execution by working on only the limited feature set than the entire row.

Let's say a frame has 4 columns named *a*, *b*, *c* and *d* and we want to add a new column with value from column *a* multiplied by value in column *b* and call it *a_times_b*. In the example below, `columns_accessed` is a list with column names *a* and *b*.

```
>>> my_frame.add_columns(lambda row: row.a * row.b, ("a_times_b", float32), columns_accessed=["a
```

`add_columns` would fail if `columns_accessed` parameter is not populated with the correct list of accessed columns. If not specified, `columns_accessed` defaults to `None` which implies that all columns might be accessed by the UDF.

More information on a row UDF can be found at [Python User Functions](#)

25.2.3 *VertexFrame* `add_vertices`

add_vertices (*self*, *source_frame*, *id_column_name*, *column_names=None*)

Add vertices to a graph.

Parameters **source_frame** : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

Frame that will be the source of the vertex data.

id_column_name : unicode

Column name for a unique id for each vertex.

column_names : list (default=`None`)

Column names that will be turned into properties for each vertex.

Returns : `_Unit`

Includes appending to a list of existing vertices.

25.2.4 *VertexFrame* `assign_sample`

assign_sample (*self*, *sample_percentages*, *sample_labels=None*, *output_column=None*, *random_seed=None*)

Randomly group rows into user-defined classes.

Parameters **sample_percentages** : list

Entries are non-negative and sum to 1. (See the note below.) If the i 'th entry of the list is p , then each row receives label i with independent probability p .

sample_labels : list (default=`None`)

Names to be used for the split classes. Defaults "TR", "TE", "VA" when the length of *sample_percentages* is 3, and defaults to `Sample_0`, `Sample_1`, ... otherwise.

output_column : unicode (default=`None`)

Name of the new column which holds the labels generated by the function.

random_seed : int32 (default=`None`)

Random seed used to generate the labels. Defaults to 0.

Returns : `_Unit`

Randomly assign classes to rows given a vector of percentages. The table receives an additional column that contains a random label. The random label is generated by a probability distribution function. The distribution function is specified by the `sample_percentages`, a list of floating point values, which add up to 1. The labels are non-negative integers drawn from the range $[0, \text{len}(S) - 1]$ where S is the `sample_percentages`. Optionally, the user can specify a list of strings to be used as the labels. If the number of labels is 3, the labels will default to “TR”, “TE” and “VA”.

Notes

The sample percentages provided by the user are preserved to at least eight decimal places, but beyond this there may be small changes due to floating point imprecision.

In particular:

- 1.The engine validates that the sum of probabilities sums to 1.0 within eight decimal places and returns an error if the sum falls outside of this range.
- 2.The probability of the final class is clamped so that each row receives a valid label with probability one.

25.2.5 *VertexFrame* `bin_column`

bin_column (*self*, *column_name*, *cutoffs*, *include_lowest=None*, *strict_binning=None*,
bin_column_name=None)
Classify data into user-defined groups.

Parameters **column_name** : unicode

Name of the column to bin.

cutoffs : list

Array of values containing bin cutoff points. Array can be list or tuple. Array values must be progressively increasing. All bin boundaries must be included, so, with N bins, you need $N+1$ values.

include_lowest : bool (default=None)

Specify how the boundary conditions are handled. True indicates that the lower bound of the bin is inclusive. False indicates that the upper bound is inclusive. Default is True.

strict_binning : bool (default=None)

Specify how values outside of the cutoffs array should be binned. If set to True, each value less than `cutoffs[0]` or greater than `cutoffs[-1]` will be assigned a bin value of -1. If set to False, values less than `cutoffs[0]` will be included in the first bin while values greater than `cutoffs[-1]` will be included in the final bin.

bin_column_name : unicode (default=None)

The name for the new binned column. Default is `<column_name>_binned`.

Returns : `_Unit`

Summarize rows of data based on the value in a single column by sorting them into bins, or groups, based on a list of bin cutoff points.

Notes

- 1.Unicode in column names is not supported and will likely cause the `drop_frames()` method (and others) to fail!
- 2.Bins IDs are 0-index: the lowest bin number is 0.

3. The first and last cutoffs are always included in the bins. When `include_lowest` is `True`, the last bin includes both cutoffs. When `include_lowest` is `False`, the first bin (bin 0) includes both cutoffs.

25.2.6 *VertexFrame* `bin_column_equal_depth`

`bin_column_equal_depth` (*self*, *column_name*, *num_bins=None*, *bin_column_name=None*)

Classify column into groups with the same frequency.

Parameters `column_name` : unicode

The column whose values are to be binned.

`num_bins` : int32 (default=None)

The maximum number of bins. Default is the Square-root choice $\lfloor \sqrt{m} \rfloor$, where m is the number of rows.

`bin_column_name` : unicode (default=None)

The name for the new column holding the grouping labels. Default is `<column_name>_binned`.

Returns : dict

A list containing the edges of each bin.

Group rows of data based on the value in a single column and add a label to identify grouping.

Equal depth binning attempts to label rows such that each bin contains the same number of elements. For n bins of a column C of length m , the bin number is determined by:

$$\left\lceil n * \frac{f(C)}{m} \right\rceil$$

where f is a tie-adjusted ranking function over values of C . If there are multiples of the same value in C , then their tie-adjusted rank is the average of their ordered rank values.

Notes

1. Unicode in column names is not supported and will likely cause the `drop_frames()` method (and others) to fail!
2. The `num_bins` parameter is considered to be the maximum permissible number of bins because the data may dictate fewer bins. For example, if the column to be binned has a quantity of X elements with only 2 distinct values and the `num_bins` parameter is greater than 2, then the actual number of bins will only be 2. This is due to a restriction that elements with an identical value must belong to the same bin.

25.2.7 *VertexFrame* `bin_column_equal_width`

`bin_column_equal_width` (*self*, *column_name*, *num_bins=None*, *bin_column_name=None*)

Classify column into same-width groups.

Parameters `column_name` : unicode

The column whose values are to be binned.

`num_bins` : int32 (default=None)

The maximum number of bins. Default is the Square-root choice $\lfloor \sqrt{m} \rfloor$, where m is the number of rows.

bin_column_name : unicode (default=None)

The name for the new column holding the grouping labels. Default is `<column_name>_binned`.

Returns : dict

A list of the edges of each bin.

Group rows of data based on the value in a single column and add a label to identify grouping.

Equal width binning places column values into groups such that the values in each group fall within the same interval and the interval width for each group is equal.

Notes

1. Unicode in column names is not supported and will likely cause the `drop_frames()` method (and others) to fail!
2. The `num_bins` parameter is considered to be the maximum permissible number of bins because the data may dictate fewer bins. For example, if the column to be binned has 10 elements with only 2 distinct values and the `num_bins` parameter is greater than 2, then the number of actual number of bins will only be 2. This is due to a restriction that elements with an identical value must belong to the same bin.

25.2.8 *VertexFrame* categorical_summary

categorical_summary (*self*, *column_inputs=None*)

[ALPHA] Compute a summary of the data in a column(s) for categorical or numerical data types.

Parameters **column_inputs** : str | tuple(str, dict) (default=None)

Comma-separated column names to summarize or tuple containing column name and dictionary of optional parameters. Optional parameters (see below for details): `top_k` (default = 10), `threshold` (default = 0.0)

Returns : dict

Summary for specified column(s) consisting of levels with their frequency and percentage

The returned value is a Map containing categorical summary for each specified column.

For each column, levels which satisfy the top k and/or threshold cutoffs are displayed along with their frequency and percentage occurrence with respect to the total rows in the dataset.

Missing data is reported when a column value is empty ("") or null.

All remaining data is grouped together in the Other category and its frequency and percentage are reported as well.

User must specify the column name and can optionally specify `top_k` and/or `threshold`.

Optional parameters:

top_k Displays levels which are in the top k most frequently occurring values for that column.

threshold Displays levels which are above the threshold percentage with respect to the total row count.

top_k and threshold Performs level pruning first based on top k and then filters out levels which satisfy the threshold criterion.

defaults Displays all levels which are in Top 10.

Examples

```
>>> frame.categorical_summary('source', 'target')
>>> frame.categorical_summary(('source', {'top_k' : 2}))
>>> frame.categorical_summary(('source', {'threshold' : 0.5}))
>>> frame.categorical_summary(('source', {'top_k' : 2}), ('target',
... {'threshold' : 0.5}))
```

Sample output (for last example above):

```
>>> {u'categorical_summary': [{u'column': u'source', u'levels': [
... {u'percentage': 0.32142857142857145, u'frequency': 9, u'level': u'thing'},
... {u'percentage': 0.32142857142857145, u'frequency': 9, u'level': u'abstraction'},
... {u'percentage': 0.25, u'frequency': 7, u'level': u'physical_entity'},
... {u'percentage': 0.10714285714285714, u'frequency': 3, u'level': u'entity'},
... {u'percentage': 0.0, u'frequency': 0, u'level': u'Missing'},
... {u'percentage': 0.0, u'frequency': 0, u'level': u'Other'}]],
... {u'column': u'target', u'levels': [
... {u'percentage': 0.07142857142857142, u'frequency': 2, u'level': u'thing'},
... {u'percentage': 0.07142857142857142, u'frequency': 2,
... u'level': u'physical_entity'},
... {u'percentage': 0.07142857142857142, u'frequency': 2, u'level': u'entity'},
... {u'percentage': 0.03571428571428571, u'frequency': 1, u'level': u'variable'},
... {u'percentage': 0.03571428571428571, u'frequency': 1, u'level': u'unit'},
... {u'percentage': 0.03571428571428571, u'frequency': 1, u'level': u'substance'},
... {u'percentage': 0.03571428571428571, u'frequency': 1, u'level': u'subject'},
... {u'percentage': 0.03571428571428571, u'frequency': 1, u'level': u'set'},
... {u'percentage': 0.03571428571428571, u'frequency': 1, u'level': u'reservoir'},
... {u'percentage': 0.03571428571428571, u'frequency': 1, u'level': u'relation'},
... {u'percentage': 0.0, u'frequency': 0, u'level': u'Missing'},
... {u'percentage': 0.5357142857142857, u'frequency': 15, u'level': u'Other'}]]]}
```

25.2.9 VertexFrame classification_metrics

classification_metrics (*self*, *label_column*, *pred_column*, *pos_label*=None, *beta*=None)

Model statistics of accuracy, precision, and others.

Parameters *label_column* : unicode

The name of the column containing the correct label for each instance.

pred_column : unicode

The name of the column containing the predicted label for each instance.

pos_label : None (default=None)

beta : float64 (default=None)

This is the beta value to use for F_β measure (default F1 measure is computed); must be greater than zero. Defaults is 1.

Returns : dict

object <object>.accuracy : double <object>.confusion_matrix : table
<object>.f_measure : double <object>.precision : double <object>.recall : double

Calculate the accuracy, precision, confusion_matrix, recall and F_β measure for a classification model.

- The **f_measure** result is the F_β measure for a classification model. The F_β measure of a binary classification model is the harmonic mean of precision and recall. If we let:

–beta $\equiv \beta$,

– T_P denotes the number of true positives,

– F_P denotes the number of false positives, and

– F_N denotes the number of false negatives

then:

$$F_\beta = (1 + \beta^2) * \frac{\frac{T_P}{T_P + F_P} * \frac{T_P}{T_P + F_N}}{\beta^2 * \frac{T_P}{T_P + F_P} + \frac{T_P}{T_P + F_N}}$$

The F_β measure for a multi-class classification model is computed as the weighted average of the F_β measure for each label, where the weight is the number of instances of each label. The determination of binary vs. multi-class is automatically inferred from the data.

- The **recall** result of a binary classification model is the proportion of positive instances that are correctly identified. If we let T_P denote the number of true positives and F_N denote the number of false negatives, then the model recall is given by $\frac{T_P}{T_P + F_N}$.

For multi-class classification models, the recall measure is computed as the weighted average of the recall for each label, where the weight is the number of instances of each label. The determination of binary vs. multi-class is automatically inferred from the data.

- The **precision** of a binary classification model is the proportion of predicted positive instances that are correctly identified. If we let T_P denote the number of true positives and F_P denote the number of false positives, then the model precision is given by: $\frac{T_P}{T_P + F_P}$.

For multi-class classification models, the precision measure is computed as the weighted average of the precision for each label, where the weight is the number of instances of each label. The determination of binary vs. multi-class is automatically inferred from the data.

- The **accuracy** of a classification model is the proportion of predictions that are correctly identified. If we let T_P denote the number of true positives, T_N denote the number of true negatives, and K denote the total number of classified instances, then the model accuracy is given by: $\frac{T_P + T_N}{K}$.

This measure applies to binary and multi-class classifiers.

- The **confusion_matrix** result is a confusion matrix for a binary classifier model, formatted for human readability.

Notes

The **confusion_matrix** is not yet implemented for multi-class classifiers.

25.2.10 *VertexFrame* column_median

column_median (*self*, *data_column*, *weights_column=None*)

Calculate the (weighted) median of a column.

Parameters **data_column** : unicode

The column whose median is to be calculated.

weights_column : unicode (default=None)

The column that provides weights (frequencies) for the median calculation. Must contain numerical data. Default is all items have a weight of 1.

Returns : dict

varies The median of the values. If a weight column is provided and no weights are finite numbers greater than 0, None is returned. The type of the median returned is the same as the contents of the data column, so a column of Longs will result in a Long median and a column of Floats will result in a Float median.

The median is the least value X in the range of the distribution so that the cumulative weight of values strictly below X is strictly less than half of the total weight and the cumulative weight of values up to and including X is greater than or equal to one-half of the total weight.

All data elements of weight less than or equal to 0 are excluded from the calculation, as are all data elements whose weight is NaN or infinite. If a weight column is provided and no weights are finite numbers greater than 0, None is returned.

25.2.11 *VertexFrame* column_mode

column_mode (*self*, *data_column*, *weights_column=None*, *max_modes_returned=None*)

Evaluate the weights assigned to rows.

Parameters **data_column** : unicode

Name of the column supplying the data.

weights_column : unicode (default=None)

Name of the column supplying the weights. Default is all items have weight of 1.

max_modes_returned : int32 (default=None)

Maximum number of modes returned. Default is 1.

Returns : dict

dict Dictionary containing summary statistics. The data returned is composed of multiple components:

mode [A mode is a data element of maximum net weight.] A set of modes is returned. The empty set is returned when the sum of the weights is 0. If the number of modes is less than or equal to the parameter *max_modes_returned*, then all modes of the data are returned. If the number of modes is greater than the *max_modes_returned* parameter, only the first *max_modes_returned* many modes (per a canonical ordering) are returned.

weight_of_mode [Weight of a mode.] If there are no data elements of finite weight greater than 0, the weight of the mode is 0. If no weights column is given, this is the number of appearances of each mode.

total_weight [Sum of all weights in the weight column.] This is the row count if no weights are given. If no weights column is given, this is the number of rows in the table with non-zero weight.

mode_count [The number of distinct modes in the data.] In the case that the data is very multimodal, this number may exceed `max_modes_returned`.

Calculate the modes of a column. A mode is a data element of maximum weight. All data elements of weight less than or equal to 0 are excluded from the calculation, as are all data elements whose weight is NaN or infinite. If there are no data elements of finite weight greater than 0, no mode is returned.

Because data distributions often have multiple modes, it is possible for a set of modes to be returned. By default, only one is returned, but by setting the optional parameter `max_modes_returned`, a larger number of modes can be returned.

25.2.12 *VertexFrame* column_names

column_names

Column identifications in the current frame.

Parameters

Returns : list

list of names of all the frame's columns

Given a Frame object, *my_frame* accessing a frame. To get the column names:

```
>>> my_columns = my_frame.column_names
>>> print my_columns
```

Now, given there are three columns *col1*, *col2*, and *col3*, the result is:

```
["col1", "col2", "col3"]
```

25.2.13 *VertexFrame* column_summary_statistics

column_summary_statistics (*self*, *data_column*, *weights_column=None*,
use_population_variance=None)

Calculate multiple statistics for a column.

Parameters **data_column** : unicode

The column to be statistically summarized. Must contain numerical data; all NaNs and infinite values are excluded from the calculation.

weights_column : unicode (default=None)

Name of column holding weights of column values.

use_population_variance : bool (default=None)

If true, the variance is calculated as the population variance. If false, the variance calculated as the sample variance. Because this option affects the variance, it affects the standard deviation and the confidence intervals as well. Default is false.

Returns : dict

dict Dictionary containing summary statistics. The data returned is composed of multiple components:

mean [[double | None]] Arithmetic mean of the data.

geometric_mean [[double | None]] Geometric mean of the data. None when there is a data element ≤ 0 , 1.0 when there are no data elements.

variance [[double | None]] None when there are ≤ 1 many data elements. Sample variance is the weighted sum of the squared distance of each data element from the weighted mean, divided by the total weight minus 1. None when the sum of the weights is ≤ 1 . Population variance is the weighted sum of the squared distance of each data element from the weighted mean, divided by the total weight.

standard_deviation [[double | None]] The square root of the variance. None when sample variance is being used and the sum of weights is ≤ 1 .

total_weight [long] The count of all data elements that are finite numbers. (In other words, after excluding NaNs and infinite values.)

minimum [[double | None]] Minimum value in the data. None when there are no data elements.

maximum [[double | None]] Maximum value in the data. None when there are no data elements.

mean_confidence_lower [[double | None]] Lower limit of the 95% confidence interval about the mean. Assumes a Gaussian distribution. None when there are no elements of positive weight.

mean_confidence_upper [[double | None]] Upper limit of the 95% confidence interval about the mean. Assumes a Gaussian distribution. None when there are no elements of positive weight.

bad_row_count [[double | None]] The number of rows containing a NaN or infinite value in either the data or weights column.

good_row_count [[double | None]] The number of rows not containing a NaN or infinite value in either the data or weights column.

positive_weight_count [[double | None]] The number of valid data elements with weight > 0 . This is the number of entries used in the statistical calculation.

non_positive_weight_count [[double | None]] The number valid data elements with finite weight ≤ 0 .

Notes

Sample Variance Sample Variance is computed by the following formula:

$$\left(\frac{1}{W - 1} \right) * \sum_i (x_i - M)^2$$

where W is sum of weights over valid elements of positive weight, and M is the weighted mean.

Population Variance Population Variance is computed by the following formula:

$$\left(\frac{1}{W}\right) * \sum_i (x_i - M)^2$$

where W is sum of weights over valid elements of positive weight, and M is the weighted mean.

Standard Deviation The square root of the variance.

Logging Invalid Data A row is bad when it contains a NaN or infinite value in either its data or weights column. In this case, it contributes to `bad_row_count`; otherwise it contributes to good row count.

A good row can be skipped because the value in its weight column is less than or equal to 0. In this case, it contributes to `non_positive_weight_count`, otherwise (when the weight is greater than 0) it contributes to `valid_data_weight_pair_count`.

Equations `bad_row_count + good_row_count = # rows in the frame`

`positive_weight_count + non_positive_weight_count = good_row_count`

In particular, when no weights column is provided and all weights are 1.0,

`non_positive_weight_count = 0` and `positive_weight_count = good_row_count`

25.2.14 *VertexFrame* `compute_misplaced_score`

`compute_misplaced_score` (*self*, *gravity*)

Parameters *gravity* : float64

Similarity measure for computing tension between 2 connected items

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.`AtkEntityType` object at 0x7f6675773090>>

25.2.15 *VertexFrame* `copy`

`copy` (*self*, *columns=None*, *where=None*, *name=None*)

Create new frame from current frame.

Parameters *columns* : str | list of str | dict (default=None)

If not None, the copy will only include the columns specified. If dict, the string pairs represent a column renaming, {`source_column_name`: `destination_column_name`}

where : function (default=None)

If not None, only those rows for which the UDF evaluates to True will be copied.

name : str (default=None)

Name of the copied frame

Returns : Frame

A new Frame of the copied data.

Copy frame or certain frame columns entirely or filtered. Useful for frame query.

Examples

Build a Frame from a csv file with 5 million rows of data; call the frame “cust”:

```
>>> my_frame = ta.Frame(source="my_data.csv")
>>> my_frame.name("cust")
```

Given the frame has columns *id*, *name*, *hair*, and *shoe*. Copy it to a new frame:

```
>>> your_frame = my_frame.copy()
```

Now we have two frames of data, each with 5 million rows. Checking the names:

```
>>> print my_frame.name()
>>> print your_frame.name()
```

Gives the results:

```
"cust"
"frame_75401b7435d7132f5470ba35..."
```

Now, let’s copy *some* of the columns from the original frame:

```
>>> our_frame = my_frame.copy(['id', 'hair'])
```

Our new frame now has two columns, *id* and *hair*, and has 5 million rows. Let’s try that again, but this time change the name of the *hair* column to *color*:

```
>>> last_frame = my_frame.copy(('id': 'id', 'hair': 'color'))
```

25.2.16 VertexFrame correlation

correlation (*self*, *data_column_names*)

Calculate correlation for two columns of current frame.

Parameters *data_column_names* : list

The names of 2 columns from which to compute the correlation.

Returns : dict

Pearson correlation coefficient of the two columns.

This method applies only to columns containing numerical data.

25.2.17 VertexFrame correlation_matrix

correlation_matrix (*self*, *data_column_names*, *matrix_name=None*)

Calculate correlation matrix for two or more columns.

Parameters *data_column_names* : list

The names of the columns from which to compute the matrix.

matrix_name : unicode (default=None)

The name for the returned matrix Frame.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A Frame with the matrix of the correlation values for the columns.

This method applies only to columns containing numerical data.

25.2.18 *VertexFrame* count

count (*self*, *where*)

Counts the number of rows which meet given criteria.

Parameters **where** : function

UDF which evaluates a row to a boolean

Returns : int

number of rows for which the where UDF evaluated to True.

25.2.19 *VertexFrame* covariance

covariance (*self*, *data_column_names*)

Calculate covariance for exactly two columns.

Parameters **data_column_names** : list

The names of two columns from which to compute the covariance.

Returns : dict

Covariance of the two columns.

This method applies only to columns containing numerical data.

25.2.20 *VertexFrame* covariance_matrix

covariance_matrix (*self*, *data_column_names*, *matrix_name=None*)

Calculate covariance matrix for two or more columns.

Parameters **data_column_names** : list

The names of the column from which to compute the matrix. Names should refer to a single column of type vector, or two or more columns of numeric scalars.

matrix_name : unicode (default=None)

The name of the new matrix.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A matrix with the covariance values for the columns.

This function applies only to columns containing numerical data.

25.2.21 *VertexFrame* cumulative_percent

cumulative_percent (*self*, *sample_col*)

[BETA] Add column to frame with cumulative percent sum.

Parameters *sample_col* : unicode

The name of the column from which to compute the cumulative percent sum.

Returns : `_Unit`

A cumulative percent sum is computed by sequentially stepping through the rows, observing the column values and keeping track of the current percentage of the total sum accounted for at the current value.

Notes

This method applies only to columns containing numerical data. Although this method will execute for columns containing negative values, the interpretation of the result will change (for example, negative percentages).

25.2.22 *VertexFrame* cumulative_sum

cumulative_sum (*self*, *sample_col*)

[BETA] Add column to frame with cumulative percent sum.

Parameters *sample_col* : unicode

The name of the column from which to compute the cumulative sum.

Returns : `_Unit`

A cumulative sum is computed by sequentially stepping through the rows, observing the column values and keeping track of the cumulative sum for each value.

Notes

This method applies only to columns containing numerical data.

25.2.23 *VertexFrame* dot_product

dot_product (*self*, *left_column_names*, *right_column_names*, *dot_product_column_name*, *default_left_values=None*, *default_right_values=None*)

[ALPHA] Calculate dot product for each row in current frame.

Parameters *left_column_names* : list

Names of columns used to create the left vector (A) for each row. Names should refer to a single column of type vector, or two or more columns of numeric scalars.

right_column_names : list

Names of columns used to create right vector (B) for each row. Names should refer to a single column of type vector, or two or more columns of numeric scalars.

dot_product_column_name : unicode

Name of column used to store the dot product.

default_left_values : list (default=None)

Default values used to substitute null values in left vector. Default is None.

default_right_values : list (default=None)

Default values used to substitute null values in right vector. Default is None.

Returns : _Unit

Calculate the dot product for each row in a frame using values from two equal-length sequences of columns.

Dot product is computed by the following formula:

The dot product of two vectors $A = [a_1, a_2, \dots, a_n]$ and $B = [b_1, b_2, \dots, b_n]$ is $a_1 * b_1 + a_2 * b_2 + \dots + a_n * b_n$. The dot product for each row is stored in a new column in the existing frame.

Notes

If default_left_values or default_right_values are not specified, any null values will be replaced by zeros.

25.2.24 VertexFrame download

download (*self*, *n=100*, *offset=0*, *columns=None*)

Download a frame from the server into client workspace.

Parameters **n** : int (default=100)

The number of rows to download to the client

offset : int (default=0)

The number of rows to skip before copying

columns : list (default=None)

Column filter, the names of columns to be included (default is all columns)

Returns : pandas.DataFrame

A new pandas dataframe object containing the downloaded frame data

Copies an trustedanalytics Frame into a Pandas DataFrame.

Examples

Frame *my_frame* accesses a frame with millions of rows of data. Get a sample of 500 rows:

```
>>> pandas_frame = my_frame.download( 500 )
```

We now have a new frame accessed by a pandas DataFrame *pandas_frame* with a copy of the first 500 rows of the original frame.

If we use the method with an offset like:

```
>>> pandas_frame = my_frame.take( 500, 100 )
```

We end up with a new frame accessed by the pandas DataFrame *pandas_frame* again, but this time it has a copy of rows 101 to 600 of the original frame.

25.2.25 *VertexFrame* drop_columns

drop_columns (*self*, *columns*)

Remove columns from the frame.

Parameters *columns* : list

Column name OR list of column names to be removed from the frame.

Returns : *_Unit*

The data from the columns is lost.

Notes

It is not possible to delete all columns from a frame. At least one column needs to remain. If it is necessary to delete all columns, then delete the frame.

25.2.26 *VertexFrame* drop_duplicates

drop_duplicates (*self*, *unique_columns=None*)

Remove duplicate vertex rows.

Parameters *unique_columns* : None (default=None)

Returns : *_Unit*

Remove duplicate vertex rows, keeping only one vertex row per uniqueness criteria match. Edges that were connected to removed vertices are also automatically dropped.

25.2.27 *VertexFrame* drop_rows

drop_rows (*self*, *predicate*)

Erase any row in the current frame which qualifies.

Parameters *predicate* : function

UDF which evaluates a row to a boolean; rows that answer True are dropped from the Frame

Examples

For this example, `my_frame` is a `Frame` object accessing a frame with lots of data for the attributes of `lions`, `tigers`, and `ligers`. Get rid of the `lions` and `tigers`:

```
>>> my_frame.drop_rows(lambda row: row.animal_type == "lion" or
...     row.animal_type == "tiger")
```

Now the frame only has information about `ligers`.

More information on a UDF can be found at [Python User Functions](#).

25.2.28 *VertexFrame* `drop_vertices`

drop_vertices (*self*, *predicate*)

Delete rows that qualify.

Parameters **predicate** : function

UDF which evaluates a row (vertex) to a boolean; vertices that answer True are dropped from the Frame

Parameters **predicate** : UDF

UDF or *lambda* which takes a row argument and evaluates to a boolean value.

Examples

Given `VertexFrame` object `my_vertex_frame` accessing a graph with lots of data for the attributes of `lions`, `tigers`, and `ligers`. Get rid of the `lions` and `tigers`:

```
>>> my_vertex_frame.drop_vertices(lambda row:
...     row.animal_type == "lion" or
...     row.animal_type == "tiger")
```

Now the frame only has information about `ligers`.

More information on UDF can be found at [Python User Functions](#)

25.2.29 *VertexFrame* `ecdf`

ecdf (*self*, *column*, *result_frame_name=None*)

Builds new frame with columns for data and distribution.

Parameters **column** : unicode

The name of the input column containing sample.

result_frame_name : unicode (default=None)

A name for the resulting frame which is created by this operation.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A new Frame containing each distinct value in the sample and its corresponding ECDF value.

Generates the *empirical cumulative distribution* for the input column.

25.2.30 *VertexFrame* entropy

entropy (*self*, *data_column*, *weights_column=None*)

Calculate the Shannon entropy of a column.

Parameters *data_column* : unicode

The column whose entropy is to be calculated.

weights_column : unicode (default=None)

The column that provides weights (frequencies) for the entropy calculation. Must contain numerical data. Default is using uniform weights of 1 for all items.

Returns : dict

Entropy.

The data column is weighted via the weights column. All data elements of weight ≤ 0 are excluded from the calculation, as are all data elements whose weight is NaN or infinite. If there are no data elements with a finite weight greater than 0, the entropy is zero.

25.2.31 *VertexFrame* export_to_csv

export_to_csv (*self*, *folder_name*, *separator=None*, *count=None*, *offset=None*)

Write current frame to HDFS in csv format.

Parameters *folder_name* : unicode

The HDFS folder path where the files will be created.

separator : None (default=None)

count : int32 (default=None)

The number of records you want. Default, or a non-positive value, is the whole frame.

offset : int32 (default=None)

The number of rows to skip before exporting to the file. Default is zero (0).

Returns : _Unit

Export the frame to a file in csv format as a Hadoop file.

25.2.32 *VertexFrame* export_to_hbase

export_to_hbase (*self*, *table_name*, *key_column_name=None*, *family_name=None*)

Write current frame to HBase table.

Parameters `table_name` : unicode

The name of the HBase table that will contain the exported frame

key_column_name : unicode (default=None)

The name of the column to be used as row key in hbase table

family_name : unicode (default=None)

The family name of the HBase table that will contain the exported frame

Returns : `_Unit`

Table must exist in HBase. Export of Vectors is not currently supported.

25.2.33 *VertexFrame* `export_to_hive`

export_to_hive (*self*, *table_name*)

Write current frame to Hive table.

Parameters `table_name` : unicode

The name of the Hive table that will contain the exported frame

Returns : `_Unit`

Table must not exist in Hive. Export of Vectors is not currently supported.

25.2.34 *VertexFrame* `export_to_jdbc`

export_to_jdbc (*self*, *table_name*, *connector_type*=None, *url*=None, *driver_name*=None, *query*=None)

Write current frame to Jdbc table.

Parameters `table_name` : unicode

jdbc table name

connector_type : unicode (default=None)

(optional) jdbc connector type

url : unicode (default=None)

(optional) connection url (includes server name, database name, user acct and password)

driver_name : unicode (default=None)

(optional) driver name

query : unicode (default=None)

(optional) query for filtering. Not supported yet.

Returns : `_Unit`

Table will be created or appended to. Export of Vectors is not currently supported.

25.2.35 *VertexFrame* export_to_json

export_to_json (*self*, *folder_name*, *count=None*, *offset=None*)

Write current frame to HDFS in JSON format.

Parameters **folder_name** : unicode

The HDFS folder path where the files will be created.

count : int32 (default=None)

The number of records you want. Default, or a non-positive value, is the whole frame.

offset : int32 (default=None)

The number of rows to skip before exporting to the file. Default is zero (0).

Returns : _Unit

Export the frame to a file in JSON format as a Hadoop file.

25.2.36 *VertexFrame* filter

filter (*self*, *predicate*)

<Missing Doc>

Parameters **predicate** : function

UDF which evaluates a row to a boolean; vertices that answer False are dropped from the Frame

25.2.37 *VertexFrame* flatten_column

flatten_column (*self*, *column*, *delimiter=None*)

Spread data to multiple rows based on cell data.

Parameters **column** : unicode

The column to be flattened.

delimiter : unicode (default=None)

The delimiter string. Default is comma (,).

Returns : _Unit

Splits cells in the specified column into multiple rows according to a string delimiter. New rows are a full copy of the original row, but the specified column only contains one value. The original row is deleted.

25.2.38 *VertexFrame* get_error_frame

get_error_frame (*self*)

Get a frame with error recordings.

Parameters

When a frame is created, another frame is transparently created to capture parse errors.

Returns **Frame** : error frame object

A new object accessing a frame that contains the parse errors of the currently active Frame or None if no error frame exists.

25.2.39 *VertexFrame* group_by

group_by (*self*, *group_by_columns*, *aggregation_arguments=None*)

[BETA] Create summarized frame.

Parameters **group_by_columns** : list

Column name or list of column names

aggregation_arguments : dict (default=None)

Aggregation function based on entire row, and/or dictionaries (one or more) of { column name str : aggregation function(s) }.

Returns : Frame

A new frame with the results of the group_by

Creates a new frame and returns a Frame object to access it. Takes a column or group of columns, finds the unique combination of values, and creates unique rows with these column values. The other columns are combined according to the aggregation argument(s).

Notes

- Column order is not guaranteed when columns are added
- The column names created by aggregation functions in the new frame are the original column name appended with the ‘_’ character and the aggregation function. For example, if the original field is *a* and the function is *avg*, the resultant column is named *a_avg*.
- An aggregation argument of *count* results in a column named *count*.
- The aggregation function *agg.count* is the only full row aggregation function supported at this time.
- Aggregation currently supports using the following functions:
 - avg
 - count
 - count_distinct
 - max
 - min

-stdev
 -sum
 -var (see glossary *Bias vs Variance*)

Examples

For setup, we will use a Frame *my_frame* accessing a frame with a column *a*:

```
>>> my_frame.inspect()

a:str
/-----/
cat
apple
bat
cat
bat
cat
```

Create a new frame, combining similar values of column *a*, and count how many of each value is in the original frame:

```
>>> new_frame = my_frame.group_by('a', agg.count)
>>> new_frame.inspect()

a:str      count:int
/-----/
cat         3
apple       1
bat         2
```

In this example, ‘*my_frame*’ is accessing a frame with three columns, *a*, *b*, and *c*:

```
>>> my_frame.inspect()

a:int  b:str  c:float
/-----/
1      alpha  3.0
1      bravo  5.0
1      alpha  5.0
2      bravo  8.0
2      bravo 12.0
```

Create a new frame from this data, grouping the rows by unique combinations of column *a* and *b*. Average the value in *c* for each group:

```
>>> new_frame = my_frame.group_by(['a', 'b'], {'c' : agg.avg})
>>> new_frame.inspect()

a:int  b:str  c_avg:float
/-----/
1      alpha  4.0
1      bravo  5.0
2      bravo 10.0
```

For this example, we use *my_frame* with columns *a*, *c*, *d*, and *e*:

```
>>> my_frame.inspect()

  a:str   c:int   d:float e:int
/-----/
ape      1      4.0     9
ape      1      8.0     8
big      1      5.0     7
big      1      6.0     6
big      1      8.0     5
```

Create a new frame from this data, grouping the rows by unique combinations of column *a* and *c*. Count each group; for column *d* calculate the average, sum and minimum value. For column *e*, save the maximum value:

```
>>> new_frame = my_frame.group_by(['a', 'c'], agg.count,
... {'d': [agg.avg, agg.sum, agg.min], 'e': agg.max})

  a    c   count  d_avg  d_sum  d_min  e_max
str  int  int    float  float  float  int
/-----/
ape  1    2     6.0   12.0   4.0    9
big  1    3     6.333 19.0   5.0    7
```

For further examples, see [Group by \(and aggregate\)](#):.

25.2.40 VertexFrame histogram

histogram (*self*, *column_name*, *num_bins*=None, *weight_column_name*=None, *bin_type*='equalwidth')
[BETA] Compute the histogram for a column in a frame.

Parameters **column_name** : unicode

Name of column to be evaluated.

num_bins : int32 (default=None)

Number of bins in histogram. Default is Square-root choice will be used (in other words `math.floor(math.sqrt(frame.row_count))`).

weight_column_name : unicode (default=None)

Name of column containing weights. Default is all observations are weighted equally.

bin_type : unicode (default=equalwidth)

The type of binning algorithm to use: ["equalwidth"|"equaldepth"] Defaults is "equalwidth".

Returns : dict

histogram A Histogram object containing the result set. The data returned is composed of multiple components:

cutoffs [array of float] A list containing the edges of each bin.

hist [array of float] A list containing count of the weighted observations found in each bin.

density [array of float] A list containing a decimal containing the percentage of observations found in the total set per bin.

Compute the histogram of the data in a column. The returned value is a Histogram object containing 3 lists one each for: the cutoff points of the bins, size of each bin, and density of each bin.

Notes

The `num_bins` parameter is considered to be the maximum permissible number of bins because the data may dictate fewer bins. With equal depth binning, for example, if the column to be binned has 10 elements with only 2 distinct values and the `num_bins` parameter is greater than 2, then the number of actual number of bins will only be 2. This is due to a restriction that elements with an identical value must belong to the same bin.

25.2.41 *VertexFrame* inspect

inspect (*self*, *n=10*, *offset=0*, *columns=None*, *wrap=None*, *truncate=None*, *round=None*, *width=80*, *margin=None*)

Prints the frame data in readable format.

Parameters *n* : int (default=10)

The number of rows to print.

offset : int (default=0)

The number of rows to skip before printing.

columns : int (default=None)

Filter columns to be included. By default, all columns are included

wrap : int or 'stripes' (default=None)

If set to 'stripes' then inspect prints rows in stripes; if set to an integer N, rows will be printed in clumps of N columns, where the columns are wrapped

truncate : int (default=None)

If set to integer N, all strings will be truncated to length N, including a tagged ellipses

round : int (default=None)

If set to integer N, all floating point numbers will be rounded and truncated to N digits

width : int (default=80)

If set to integer N, the print out will try to honor a max line width of N

margin : int (default=None)

('stripes' mode only) If set to integer N, the margin for printing names in a stripe will be limited to N characters

Examples

Given a frame of data and a Frame to access it. To look at the first 4 rows of data:

```
>>> print my_frame.inspect(4)

column defs ->  animal:str  name:str  age:int  weight:float
               /-----/
frame data ->  human      George    8        542.5
               human      Ursula    6        495.0
```

ape	Ape	41	400.0
elephant	Shep	5	8630.0

For other examples, see *Inspect The Data*.

25.2.42 *VertexFrame* join

join (*self*, *right*, *left_on*, *right_on*=None, *how*=‘inner’, *name*=None)
[BETA] Join operation on one or two frames, creating a new frame.

Parameters **right** : Frame

Another frame to join with

left_on : str

Name of the column in the left frame used to match up the two frames.

right_on : str (default=None)

Name of the column in the right frame used to match up the two frames. Default is the same as the left frame.

how : str (default=inner)

How to qualify the data to be joined together. Must be one of the following: ‘left’, ‘right’, ‘inner’, ‘outer’. Default is ‘inner’

name : str (default=None)

Name of the result grouped frame

Returns : Frame

A new frame with the results of the join

Create a new frame from a SQL JOIN operation with another frame. The frame on the ‘left’ is the currently active frame. The frame on the ‘right’ is another frame. This method takes a column in the left frame and matches its values with a column in the right frame. Using the default ‘how’ option [‘inner’] will only allow data in the resultant frame if both the left and right frames have the same value in the matching column. Using the ‘left’ ‘how’ option will allow any data in the resultant frame if it exists in the left frame, but will allow any data from the right frame if it has a value in its column which matches the value in the left frame column. Using the ‘right’ option works similarly, except it keeps all the data from the right frame and only the data from the left frame when it matches. The ‘outer’ option provides a frame with data from both frames where the left and right frames did not have the same value in the matching column.

Notes

When a column is named the same in both frames, it will result in two columns in the new frame. The column from the *left* frame (originally the current frame) will be copied and the column name will have the string “_L” added to it. The same thing will happen with the column from the *right* frame, except its name has the string “_R” appended. The order of columns after this method is called is not guaranteed.

It is recommended that you rename the columns to meaningful terms prior to using the `join` method. Keep in mind that unicode in column names will likely cause the `drop_frames()` method (and others) to fail!

Examples

For this example, we will use a Frame *my_frame* accessing a frame with columns *a*, *b*, *c*, and a Frame *your_frame* accessing a frame with columns *a*, *d*, *e*. Join the two frames keeping only those rows having the same value in column *a*:

```
>>> print my_frame.inspect()

  a:unicode  b:unicode  c:unicode
/-----/
alligator   bear       cat
apple       berry      cantaloupe
auto        bus        car
mirror      frog       ball

>>> print your_frame.inspect()

  b:unicode  c:int  d:unicode
/-----/
berry       5218  frog
blue        0    log
bus         871  dog

>>> joined_frame = my_frame.join(your_frame, 'b', how='inner')
```

Now, *joined_frame* is a Frame accessing a frame with the columns *a*, *b*, *c_L*, *c_R*, and *d*. The data in the new frame will be from the rows where column 'a' was the same in both frames.

```
>>> print joined_frame.inspect()

  a:unicode  b:unicode  c_L:unicode  c_R:int64  d:unicode
/-----/
apple       berry      cantaloupe    5218      frog
auto        bus        car              871      dog
```

More examples can be found in the [user manual](#).

25.2.43 VertexFrame loadhbase

loadhbase (*self*, *table_name*, *schema*, *start_tag=None*, *end_tag=None*)

Append data from an hBase table into an existing (possibly empty) FrameRDD

Parameters *table_name* : unicode

hbase table name

schema : list

hbase schema as a list of tuples (columnFamily, columnName, dataType for cell value)

start_tag : unicode (default=None)

optional start tag for filtering

end_tag : unicode (default=None)

optional end tag for filtering

Returns : <bound method AtkEntityType.__name__ of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
 the initial FrameRDD with the hbase data appended

Append data from an hBase table into an existing (possibly empty) FrameRDD

25.2.44 *VertexFrame* loadhive

loadhive (*self, query*)

Append data from a hive table into an existing (possibly empty) frame

Parameters **query** : unicode

Initial query to run at load time

Returns : <bound method AtkEntityType.__name__ of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
 the initial frame with the hive data appended

Append data from a hive table into an existing (possibly empty) frame

25.2.45 *VertexFrame* loadjdbc

loadjdbc (*self, table_name, connector_type=None, url=None, driver_name=None, query=None*)

Append data from a Jdbc table into an existing (possibly empty) frame

Parameters **table_name** : unicode

table name

connector_type : unicode (default=None)

(optional) connector type

url : unicode (default=None)

(optional) connection url (includes server name, database name, user acct and password)

driver_name : unicode (default=None)

(optional) driver name

query : unicode (default=None)

(optional) query for filtering. Not supported yet.

Returns : <bound method AtkEntityType.__name__ of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
 the initial frame with the Jdbc data appended

Append data from a Jdbc table into an existing (possibly empty) frame

25.2.46 *VertexFrame* name

name

Set or get the name of the frame object.

Parameters

Change or retrieve frame object identification. Identification names must start with a letter and are limited to alphanumeric characters and the `_` character.

Examples

```
>>> my_frame.name

"csv_data"

>>> my_frame.name = "cleaned_data"
>>> my_frame.name

"cleaned_data"
```

25.2.47 *VertexFrame* quantiles

quantiles (*self*, *column_name*, *quantiles*)

New frame with Quantiles and their values.

Parameters **column_name** : unicode

The column to calculate quantiles.

quantiles : list

What is being requested.

Returns : <bound method *AtkEntityType.__name__* of

<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A new frame with two columns (float64): requested Quantiles and their respective values.

Calculate quantiles on the given column.

25.2.48 *VertexFrame* rename_columns

rename_columns (*self*, *names*)

Rename columns for vertex frame.

Parameters **names** : None

Returns : `_Unit`

25.2.49 *VertexFrame* row_count

row_count

Number of rows in the current frame.

Parameters

Returns : int

The number of rows in the frame

Get the number of rows:

```
>>> my_frame.row_count
```

The result given is:

```
81734
```

25.2.50 *VertexFrame* schema

schema

Current frame column names and types.

Parameters

Returns : list

list of tuples of the form (<column name>, <data type>)

The schema of the current frame is a list of column names and associated data types. It is retrieved as a list of tuples. Each tuple has the name and data type of one of the frame's columns.

Examples

Given that we have an existing data frame *my_data*, create a Frame, then show the frame schema:

```
>>> BF = ta.get_frame('my_data')
>>> print BF.schema
```

The result is:

```
[("col1", str), ("col2", numpy.int32)]
```

25.2.51 *VertexFrame* sort

sort (*self*, *columns*, *ascending=True*)

[BETA] Sort the data in a frame.

Parameters **columns** : str | list of str | list of tuples

Either a column name, a list of column names, or a list of tuples where each tuple is a name and an ascending bool value.

ascending : bool (default=True)

True for ascending, False for descending.

Sort a frame by column values either ascending or descending.

Examples

Sort a single column:

```
>>> frame.sort('column_name')
```

Sort a single column ascending:

```
>>> frame.sort('column_name', True)
```

Sort a single column descending:

```
>>> frame.sort('column_name', False)
```

Sort multiple columns:

```
>>> frame.sort(['col1', 'col2'])
```

Sort multiple columns ascending:

```
>>> frame.sort(['col1', 'col2'], True)
```

Sort multiple columns descending:

```
>>> frame.sort(['col1', 'col2'], False)
```

Sort multiple columns: 'col1' ascending and 'col2' descending:

```
>>> frame.sort([ ('col1', True), ('col2', False) ])
```

25.2.52 VertexFrame sorted_k

sorted_k (*self*, *k*, *column_names_and_ascending*, *reduce_tree_depth=None*)

[ALPHA] Get a sorted subset of the data.

Parameters **k** : int32

Number of sorted records to return.

column_names_and_ascending : list

Column names to sort by, and true to sort column by ascending order, or false for descending order.

reduce_tree_depth : int32 (default=None)

Advanced tuning parameter which determines the depth of the reduce-tree for the sorted_k plugin. This plugin uses Spark's treeReduce() for scalability. The default depth is 2.

Returns : <bound method AtkEntityType.__name__ of

<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A new frame with the first *k* sorted rows from the original frame.

Take the first *k* (sorted) rows for the currently active Frame. Rows are sorted by column values in either ascending or descending order.

Returning the first *k* (sorted) rows is more efficient than sorting the entire frame when *k* is much smaller than the number of rows in the frame.

Notes

The number of sorted rows (*k*) should be much smaller than the number of rows in the original frame.

In particular:

1. The number of sorted rows (*k*) returned should fit in Spark driver memory.

The maximum size of serialized results that can fit in the Spark driver is set by the Spark configuration parameter *spark.driver.maxResultSize*.

2. If you encounter a Kryo buffer overflow exception, increase the Spark

configuration parameter *spark.kryoserializer.buffer.max.mb*.

3. Use `Frame.sort()` instead if the number of sorted rows (*k*) is

very large (i.e., cannot fit in Spark driver memory).

25.2.53 *VertexFrame* status

status

Current frame life cycle status.

Parameters

Returns : str

Status of the frame

One of three statuses: Active, Deleted, Deleted_Final Active: Frame is available for use Deleted: Frame has been scheduled for deletion can be unscheduled by modifying Deleted_Final: Frame's backend files have been removed from disk.

Examples

Given that we have an existing data frame *my_data*, create a Frame, then show the frame schema:

```
>>> BF = ta.get_frame('my_data')
>>> print BF.status
```

The result is:

```
u'Active'
```

25.2.54 *VertexFrame* take

take (*self*, *n*, *offset=0*, *columns=None*)

Get data subset.

Parameters *n* : int

The number of rows to copy to the client from the frame.

offset : int (default=0)

The number of rows to skip before starting to copy

columns : str | iterable of str (default=None)

If not None, only the given columns' data will be provided. By default, all columns are included

Returns : list

A list of lists, where each contained list is the data for one row.

Take a subset of the currently active Frame.

Notes

The data is considered 'unstructured', therefore taking a certain number of rows, the rows obtained may be different every time the command is executed, even if the parameters do not change.

Examples

Frame *my_frame* accesses a frame with millions of rows of data. Get a sample of 5000 rows:

```
>>> my_data_list = my_frame.take( 5000 )
```

We now have a list of data from the original frame.

```
>>> print my_data_list

[[ 1, "text", 3.1415962 ]
 [ 2, "bob", 25.0 ]
 [ 3, "weave", .001 ]
 ...]
```

If we use the method with an offset like:

```
>>> my_data_list = my_frame.take( 5000, 1000 )
```

We end up with a new list, but this time it has a copy of the data from rows 1001 to 5000 of the original frame.

25.2.55 *VertexFrame* tally

tally (*self*, *sample_col*, *count_val*)

[BETA] Count number of times a value is seen.

Parameters `sample_col` : unicode

The name of the column from which to compute the cumulative count.

`count_val` : unicode

The column value to be used for the counts.

Returns : `_Unit`

A cumulative count is computed by sequentially stepping through the rows, observing the column values and keeping track of the the number of times the specified *count_value* has been seen.

25.2.56 *VertexFrame* tally_percent

tally_percent (*self*, *sample_col*, *count_val*)

[BETA] Compute a cumulative percent count.

Parameters `sample_col` : unicode

The name of the column from which to compute the cumulative sum.

`count_val` : unicode

The column value to be used for the counts.

Returns : `_Unit`

A cumulative percent count is computed by sequentially stepping through the rows, observing the column values and keeping track of the percentage of the total number of times the specified *count_value* has been seen up to the current value.

25.2.57 *VertexFrame* top_k

top_k (*self*, *column_name*, *k*, *weights_column*=None)

Most or least frequent column values.

Parameters `column_name` : unicode

The column whose top (or bottom) K distinct values are to be calculated.

`k` : int32

Number of entries to return (If k is negative, return bottom k).

weights_column : unicode (default=None)

The column that provides weights (frequencies) for the topK calculation. Must contain numerical data. Default is 1 for all items.

Returns : <bound method `AtkEntityType.__name__` of

<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

An object with access to the frame of data.

Calculate the top (or bottom) K distinct values by count of a column. The column can be weighted. All data elements of weight ≤ 0 are excluded from the calculation, as are all data elements whose weight is NaN or infinite. If there are no data elements of finite weight > 0 , then topK is empty.

25.2.58 *VertexFrame* unflatten_column

unflatten_column (*self*, *composite_key_column_names*, *delimiter=None*)

Compacts data from multiple rows based on cell data.

Parameters *composite_key_column_names* : list

name of the user column to be used as keys for unflattening.

delimiter : unicode (default=None)

separator for the data in the result columns. Default is comma (,).

Returns : _Unit

Groups together cells in all columns (less the composite key) using “,” as string delimiter. The original rows are deleted. The grouping takes place based on a composite key passed as arguments.

class VertexFrame

A list of Vertices owned by a Graph..

A VertexFrame is similar to a Frame but with a few important differences:

- VertexFrames are not instantiated directly by the user, instead they are created by defining a vertex type in a graph
- Each row of a VertexFrame represents a vertex in a graph
- VertexFrames have many of the same methods as Frames but not all (for example, `flatten_column()`)
- VertexFrames have extra methods not found on Frames (for example, `add_vertices()`)
- Removing a vertex (or row) from a VertexFrame also removes edges connected to that vertex from the graph
- VertexFrames have special system columns (`_vid`, `_label`) that are maintained automatically by the system and cannot be modified by the user
- VertexFrames have a special user defined id column whose value uniquely identifies the vertex
- “Columns” on a VertexFrame can also be thought of as “properties” on vertices

Attributes

<code>column_names</code>	Column identifications in the current frame.
<code>name</code>	Set or get the name of the frame object.
<code>row_count</code>	Number of rows in the current frame.
<code>schema</code>	Current frame column names and types.
<code>status</code>	Current frame life cycle status.

Methods

<code>__init__(self[, source, graph, label, _info])</code>	Examples
<code>add_columns(self, func, schema[, columns_accessed])</code>	Add columns to current frame.
<code>add_vertices(self, source_frame, id_column_name[, column_names])</code>	Add vertices to a graph.
<code>assign_sample(self, sample_percentages[, sample_labels, ...])</code>	Randomly group rows into user-defined classes.

Table 25.2 – continued from previous page

<code>bin_column(self, column_name, cutoffs[, include_lowest, strict_binning, ...])</code>	Classify data into user-defined groups.
<code>bin_column_equal_depth(self, column_name[, num_bins, ...])</code>	Classify column into groups with the same frequency.
<code>bin_column_equal_width(self, column_name[, num_bins, ...])</code>	Classify column into same-width groups.
<code>categorical_summary(self, *column_inputs)</code>	[ALPHA] Compute a summary of the data in a column.
<code>classification_metrics(self, label_column, pred_column[, ...])</code>	Model statistics of accuracy, precision, and others.
<code>column_median(self, data_column[, weights_column])</code>	Calculate the (weighted) median of a column.
<code>column_mode(self, data_column[, weights_column, max_modes_returned])</code>	Evaluate the weights assigned to rows.
<code>column_summary_statistics(self, data_column[, ...])</code>	Calculate multiple statistics for a column.
<code>compute_misplaced_score(self, gravity)</code>	
<code>copy(self[, columns, where, name])</code>	Create new frame from current frame.
<code>correlation(self, data_column_names)</code>	Calculate correlation for two columns of current frame.
<code>correlation_matrix(self, data_column_names[, matrix_name])</code>	Calculate correlation matrix for two or more columns.
<code>count(self, where)</code>	Counts the number of rows which meet given criteria.
<code>covariance(self, data_column_names)</code>	Calculate covariance for exactly two columns.
<code>covariance_matrix(self, data_column_names[, matrix_name])</code>	Calculate covariance matrix for two or more columns.
<code>cumulative_percent(self, sample_col)</code>	[BETA] Add column to frame with cumulative percent.
<code>cumulative_sum(self, sample_col)</code>	[BETA] Add column to frame with cumulative percent.
<code>dot_product(self, left_column_names, right_column_names, ...[, ...])</code>	[ALPHA] Calculate dot product for each row in current frame.
<code>download(self[, n, offset, columns])</code>	Download a frame from the server into client workspace.
<code>drop_columns(self, columns)</code>	Remove columns from the frame.
<code>drop_duplicates(self[, unique_columns])</code>	Remove duplicate vertex rows.
<code>drop_rows(self, predicate)</code>	Erase any row in the current frame which qualifies.
<code>drop_vertices(self, predicate)</code>	Delete rows that qualify.
<code>ecdf(self, column[, result_frame_name])</code>	Builds new frame with columns for data and distribution.
<code>entropy(self, data_column[, weights_column])</code>	Calculate the Shannon entropy of a column.
<code>export_to_csv(self, folder_name[, separator, count, offset])</code>	Write current frame to HDFS in csv format.
<code>export_to_hbase(self, table_name[, key_column_name, family_name])</code>	Write current frame to HBase table.
<code>export_to_hive(self, table_name)</code>	Write current frame to Hive table.
<code>export_to_jdbc(self, table_name[, connector_type, url, driver_name, ...])</code>	Write current frame to Jdbc table.
<code>export_to_json(self, folder_name[, count, offset])</code>	Write current frame to HDFS in JSON format.
<code>filter(self, predicate)</code>	<Missing Doc>
<code>flatten_column(self, column[, delimiter])</code>	Spread data to multiple rows based on cell data.
<code>get_error_frame(self)</code>	Get a frame with error recordings.
<code>group_by(self, group_by_columns, *aggregation_arguments)</code>	[BETA] Create summarized frame.
<code>histogram(self, column_name[, num_bins, weight_column_name, bin_type])</code>	[BETA] Compute the histogram for a column in a frame.
<code>inspect(self[, n, offset, columns, wrap, truncate, round, width, margin])</code>	Prints the frame data in readable format.
<code>join(self, right, left_on[, right_on, how, name])</code>	[BETA] Join operation on one or two frames, creating new frame.
<code>loadhbase(self, table_name, schema[, start_tag, end_tag])</code>	Append data from an hBase table into an existing (possibly new) frame.
<code>loadhive(self, query)</code>	Append data from a hive table into an existing (possibly new) frame.
<code>loadjdbc(self, table_name[, connector_type, url, driver_name, query])</code>	Append data from a Jdbc table into an existing (possibly new) frame.
<code>quantiles(self, column_name, quantiles)</code>	New frame with Quantiles and their values.
<code>rename_columns(self, names)</code>	Rename columns for vertex frame.
<code>sort(self, columns[, ascending])</code>	[BETA] Sort the data in a frame.
<code>sorted_k(self, k, column_names_and_ascending[, reduce_tree_depth])</code>	[ALPHA] Get a sorted subset of the data.
<code>take(self, n[, offset, columns])</code>	Get data subset.
<code>tally(self, sample_col, count_val)</code>	[BETA] Count number of times a value is seen.
<code>tally_percent(self, sample_col, count_val)</code>	[BETA] Compute a cumulative percent count.
<code>top_k(self, column_name, k[, weights_column])</code>	Most or least frequent column values.
<code>unflatten_column(self, composite_key_column_names[, delimiter])</code>	Compacts data from multiple rows based on cell data.

```
__init__(self, source=None, graph=None, label=None)
```

Examples

Parameters `source` : (default=None)

graph : (default=None)

label : (default=None)

Given a data file, create a frame, move the data to graph and then define a new VertexFrame and add data to it:

Retrieve a previously defined graph and retrieve a VertexFrame from it:

```
>>> my_graph = ta.get_graph("your_graph")
>>> my_vertex_frame = my_graph.vertices["your_label"]
```

Calling methods on a VertexFrame:

```
>>> my_vertex_frame.vertices["your_label"].inspect(20)
```

Convert a VertexFrame to a frame:

```
>>> new_Frame = my_vertex_frame.vertices["label"].copy()
```

25.3 Frames Frame

25.3.1 Frame `__init__`

```
__init__(self, source=None, name=None)
```

Create a Frame/frame.

Parameters `source` : CsvFile | Frame (default=None)

A source of initial data.

name : str (default=None)

The name of the newly created frame. Default is None.

Notes

A frame with no name is subject to garbage collection.

If a string in the CSV file starts and ends with a double-quote (") character, the character is stripped off of the data before it is put into the field. Anything, including delimiters, between the double-quote characters is considered part of the str. If the first character after the delimiter is anything other than a double-quote character, the string will be composed of all the characters between the delimiters, including double-quotes. If the first field type is str, leading spaces on each row are considered part of the str. If the last field type is str, trailing spaces on each row are considered part of the str.

Examples

Create a new frame based upon the data described in the `CsvFile` object `my_csv_schema`. Name the frame “myframe”. Create a Frame `my_frame` to access the data:

```
>>> my_frame = ta.Frame(my_csv_schema, "myframe")
```

A Frame object has been created and `my_frame` is its proxy. It brought in the data described by `my_csv_schema`. It is named `myframe`.

Create an empty frame; name it “yourframe”:

```
>>> your_frame = ta.Frame(name='yourframe')
```

A frame has been created and Frame `your_frame` is its proxy. It has no data yet, but it does have the name `yourframe`.

25.3.2 Frame `add_columns`

`add_columns` (*self, func, schema, columns_accessed=None*)

Add columns to current frame.

Parameters **func** : UDF

User-Defined Function (UDF) which takes the values in the row and produces a value, or collection of values, for the new cell(s).

schema : tuple | list of tuples

The schema for the results of the UDF, indicating the new column(s) to add. Each tuple provides the column name and data type, and is of the form (str, type).

columns_accessed : list (default=None)

List of columns which the UDF will access. This adds significant performance benefit if we know which column(s) will be needed to execute the UDF, especially when the frame has significantly more columns than those being used to evaluate the UDF.

Assigns data to column based on evaluating a function for each row.

Notes

- 1.The row UDF (‘func’) must return a value in the same format as specified by the schema. See [Python User Functions](#).
- 2.Unicode in column names is not supported and will likely cause the `drop_frames()` method (and others) to fail!

Examples

Given a Frame `my_frame` identifying a data frame with two int32 columns `column1` and `column2`. Add a third column `column3` as an int32 and fill it with the contents of `column1` and `column2` multiplied together:

```
>>> my_frame.add_columns(lambda row: row.column1*row.column2,
... ('column3', int32))
```

The frame now has three columns, *column1*, *column2* and *column3*. The type of *column3* is an int32, and the value is the product of *column1* and *column2*.

Add a string column *column4* that is empty:

```
>>> my_frame.add_columns(lambda row: '', ('column4', str))
```

The Frame object *my_frame* now has four columns *column1*, *column2*, *column3*, and *column4*. The first three columns are int32 and the fourth column is str. Column *column4* has an empty string ('') in every row.

Multiple columns can be added at the same time. Add a column *a_times_b* and fill it with the contents of column *a* multiplied by the contents of column *b*. At the same time, add a column *a_plus_b* and fill it with the contents of column *a* plus the contents of column *b*:

```
>>> my_frame.add_columns(lambda row: [row.a * row.b, row.a +
... row.b], [("a_times_b", float32), ("a_plus_b", float32)])
```

Two new columns are created, “a_times_b” and “a_plus_b”, with the appropriate contents.

Given a frame of data and Frame *my_frame* points to it. In addition we have defined a UDF *func*. Run *func* on each row of the frame and put the result in a new int column *calculated_a*:

```
>>> my_frame.add_columns( func, ("calculated_a", int))
```

Now the frame has a column *calculated_a* which has been filled with the results of the UDF *func*.

A UDF must return a value in the same format as the column is defined. In most cases this is automatically the case, but sometimes it is less obvious. Given a UDF *function_b* which returns a value in a list, store the result in a new column *calculated_b*:

```
>>> my_frame.add_columns(function_b, ("calculated_b", float32))
```

This would result in an error because *function_b* is returning a value as a single element list like [2.4], but our column is defined as a tuple. The column must be defined as a list:

```
>>> my_frame.add_columns(function_b, [("calculated_b", float32)])
```

To run an optimized version of *add_columns*, *columns_accessed* parameter can be populated with the column names which are being accessed in UDF. This speeds up the execution by working on only the limited feature set than the entire row.

Let’s say a frame has 4 columns named *a*, *b*, *c* and *d* and we want to add a new column with value from column *a* multiplied by value in column *b* and call it *a_times_b*. In the example below, *columns_accessed* is a list with column names *a* and *b*.

```
>>> my_frame.add_columns(lambda row: row.a * row.b, ("a_times_b", float32), columns_accessed=["a", "b"])
```

add_columns would fail if *columns_accessed* parameter is not populated with the correct list of accessed columns. If not specified, *columns_accessed* defaults to None which implies that all columns might be accessed by the UDF.

More information on a row UDF can be found at [Python User Functions](#)

25.3.3 Frame append

append (*self*, *data*)

Adds more data to the current frame.

Parameters *data* : Data source

Data source, see [Data Sources](#)

Examples

Given a frame with a single column, *col_1*:

```
>>> my_frame.inspect(4)
col_1:str
/-----/
dog
cat
bear
donkey
```

and a frame with two columns, **col_1** and **col_2**:

..code::

```
>>> your_frame.inspect(4)
col_1:str  col_qty:int32
/-----/
bear      15
cat       2
snake     8
horse     5
```

Column *col_1* means the same thing in both frames. The Frame *my_frame* points to the first frame and *your_frame* points to the second. To add the contents of *your_frame* to *my_frame*:

```
>>> my_frame.append(your_frame)
>>> my_frame.inspect(8)
col_1:str  col_2:int32
/-----/
dog      None
bear     15
bear     None
horse    5
cat      None
cat      2
donkey   None
snake    5
```

Now the first frame has two columns, *col_1* and *col_2*. Column *col_1* has the data from *col_1* in both original frames. Column *col_2* has None (undefined) in all of the rows in the original first frame, and has the value of the second frame column, *col_2*, in the rows matching the new data in *col_1*.

Breaking it down differently, the original rows referred to by *my_frame* have a new column, *col_2*, and this new column is filled with non-defined data. The frame referred to by *your_frame*, is then added to the bottom.

25.3.4 Frame assign_sample

assign_sample(*self*, *sample_percentages*, *sample_labels=None*, *output_column=None*, *random_seed=None*)

Randomly group rows into user-defined classes.

Parameters *sample_percentages* : list

Entries are non-negative and sum to 1. (See the note below.) If the i 'th entry of the list is p , then each row receives label i with independent probability p .

sample_labels : list (default=None)

Names to be used for the split classes. Defaults "TR", "TE", "VA" when the length of *sample_percentages* is 3, and defaults to Sample_0, Sample_1, ... otherwise.

output_column : unicode (default=None)

Name of the new column which holds the labels generated by the function.

random_seed : int32 (default=None)

Random seed used to generate the labels. Defaults to 0.

Returns : _Unit

Randomly assign classes to rows given a vector of percentages. The table receives an additional column that contains a random label. The random label is generated by a probability distribution function. The distribution function is specified by the *sample_percentages*, a list of floating point values, which add up to 1. The labels are non-negative integers drawn from the range $[0, \text{len}(S) - 1]$ where S is the *sample_percentages*. Optionally, the user can specify a list of strings to be used as the labels. If the number of labels is 3, the labels will default to "TR", "TE" and "VA".

Notes

The sample percentages provided by the user are preserved to at least eight decimal places, but beyond this there may be small changes due to floating point imprecision.

In particular:

- 1.The engine validates that the sum of probabilities sums to 1.0 within eight decimal places and returns an error if the sum falls outside of this range.
- 2.The probability of the final class is clamped so that each row receives a valid label with probability one.

25.3.5 Frame bin_column

bin_column (*self*, *column_name*, *cutoffs*, *include_lowest=None*, *strict_binning=None*, *bin_column_name=None*)

Classify data into user-defined groups.

Parameters **column_name** : unicode

Name of the column to bin.

cutoffs : list

Array of values containing bin cutoff points. Array can be list or tuple. Array values must be progressively increasing. All bin boundaries must be included, so, with N bins, you need $N+1$ values.

include_lowest : bool (default=None)

Specify how the boundary conditions are handled. True indicates that the lower bound of the bin is inclusive. False indicates that the upper bound is inclusive. Default is True.

strict_binning : bool (default=None)

Specify how values outside of the cutoffs array should be binned. If set to `True`, each value less than `cutoffs[0]` or greater than `cutoffs[-1]` will be assigned a bin value of `-1`. If set to `False`, values less than `cutoffs[0]` will be included in the first bin while values greater than `cutoffs[-1]` will be included in the final bin.

bin_column_name : unicode (default=`None`)

The name for the new binned column. Default is `<column_name>_binned`.

Returns : `_Unit`

Summarize rows of data based on the value in a single column by sorting them into bins, or groups, based on a list of bin cutoff points.

Notes

1. Unicode in column names is not supported and will likely cause the `drop_frames()` method (and others) to fail!
2. Bins IDs are 0-index: the lowest bin number is 0.
3. The first and last cutoffs are always included in the bins. When `include_lowest` is `True`, the last bin includes both cutoffs. When `include_lowest` is `False`, the first bin (bin 0) includes both cutoffs.

25.3.6 *Frame* `bin_column_equal_depth`

bin_column_equal_depth (*self*, *column_name*, *num_bins=None*, *bin_column_name=None*)

Classify column into groups with the same frequency.

Parameters **column_name** : unicode

The column whose values are to be binned.

num_bins : int32 (default=`None`)

The maximum number of bins. Default is the Square-root choice $\lfloor \sqrt{m} \rfloor$, where m is the number of rows.

bin_column_name : unicode (default=`None`)

The name for the new column holding the grouping labels. Default is `<column_name>_binned`.

Returns : dict

A list containing the edges of each bin.

Group rows of data based on the value in a single column and add a label to identify grouping.

Equal depth binning attempts to label rows such that each bin contains the same number of elements. For n bins of a column C of length m , the bin number is determined by:

$$\lceil n * \frac{f(C)}{m} \rceil$$

where f is a tie-adjusted ranking function over values of C . If there are multiples of the same value in C , then their tie-adjusted rank is the average of their ordered rank values.

Notes

1. Unicode in column names is not supported and will likely cause the `drop_frames()` method (and others) to fail!

2. The `num_bins` parameter is considered to be the maximum permissible number of bins because the data may dictate fewer bins. For example, if the column to be binned has a quantity of X elements with only 2 distinct values and the `num_bins` parameter is greater than 2, then the actual number of bins will only be 2. This is due to a restriction that elements with an identical value must belong to the same bin.

25.3.7 *Frame* `bin_column_equal_width`

`bin_column_equal_width` (*self*, *column_name*, *num_bins=None*, *bin_column_name=None*)

Classify column into same-width groups.

Parameters `column_name` : unicode

The column whose values are to be binned.

`num_bins` : int32 (default=None)

The maximum number of bins. Default is the Square-root choice $\lfloor \sqrt{m} \rfloor$, where m is the number of rows.

`bin_column_name` : unicode (default=None)

The name for the new column holding the grouping labels. Default is `<column_name>_binned`.

Returns : dict

A list of the edges of each bin.

Group rows of data based on the value in a single column and add a label to identify grouping.

Equal width binning places column values into groups such that the values in each group fall within the same interval and the interval width for each group is equal.

Notes

1. Unicode in column names is not supported and will likely cause the `drop_frames()` method (and others) to fail!
2. The `num_bins` parameter is considered to be the maximum permissible number of bins because the data may dictate fewer bins. For example, if the column to be binned has 10 elements with only 2 distinct values and the `num_bins` parameter is greater than 2, then the number of actual number of bins will only be 2. This is due to a restriction that elements with an identical value must belong to the same bin.

25.3.8 *Frame* `categorical_summary`

`categorical_summary` (*self*, *column_inputs=None*)

[ALPHA] Compute a summary of the data in a column(s) for categorical or numerical data types.

Parameters `column_inputs` : str | tuple(str, dict) (default=None)

Comma-separated column names to summarize or tuple containing column name and dictionary of optional parameters. Optional parameters (see below for details): `top_k` (default = 10), `threshold` (default = 0.0)

Returns : dict

Summary for specified column(s) consisting of levels with their frequency and percentage

The returned value is a Map containing categorical summary for each specified column.

For each column, levels which satisfy the top k and/or threshold cutoffs are displayed along with their frequency and percentage occurrence with respect to the total rows in the dataset.

Missing data is reported when a column value is empty ("") or null.

All remaining data is grouped together in the Other category and its frequency and percentage are reported as well.

User must specify the column name and can optionally specify top_k and/or threshold.

Optional parameters:

top_k Displays levels which are in the top k most frequently occurring values for that column.

threshold Displays levels which are above the threshold percentage with respect to the total row count.

top_k and threshold Performs level pruning first based on top k and then filters out levels which satisfy the threshold criterion.

defaults Displays all levels which are in Top 10.

Examples

```
>>> frame.categorical_summary('source', 'target')
>>> frame.categorical_summary(('source', {'top_k' : 2}))
>>> frame.categorical_summary(('source', {'threshold' : 0.5}))
>>> frame.categorical_summary(('source', {'top_k' : 2}), ('target',
... {'threshold' : 0.5}))
```

Sample output (for last example above):

```
>>> {u'categorical_summary': [{u'column': u'source', u'levels': [
... {u'percentage': 0.32142857142857145, u'frequency': 9, u'level': u'thing'},
... {u'percentage': 0.32142857142857145, u'frequency': 9, u'level': u'abstraction'},
... {u'percentage': 0.25, u'frequency': 7, u'level': u'physical_entity'},
... {u'percentage': 0.10714285714285714, u'frequency': 3, u'level': u'entity'},
... {u'percentage': 0.0, u'frequency': 0, u'level': u'Missing'},
... {u'percentage': 0.0, u'frequency': 0, u'level': u'Other'}]],
... {u'column': u'target', u'levels': [
... {u'percentage': 0.07142857142857142, u'frequency': 2, u'level': u'thing'},
... {u'percentage': 0.07142857142857142, u'frequency': 2,
... u'level': u'physical_entity'},
... {u'percentage': 0.07142857142857142, u'frequency': 2, u'level': u'entity'},
... {u'percentage': 0.03571428571428571, u'frequency': 1, u'level': u'variable'},
... {u'percentage': 0.03571428571428571, u'frequency': 1, u'level': u'unit'},
... {u'percentage': 0.03571428571428571, u'frequency': 1, u'level': u'substance'},
... {u'percentage': 0.03571428571428571, u'frequency': 1, u'level': u'subject'},
... {u'percentage': 0.03571428571428571, u'frequency': 1, u'level': u'set'},
... {u'percentage': 0.03571428571428571, u'frequency': 1, u'level': u'reservoir'},
... {u'percentage': 0.03571428571428571, u'frequency': 1, u'level': u'relation'},
... {u'percentage': 0.0, u'frequency': 0, u'level': u'Missing'},
... {u'percentage': 0.5357142857142857, u'frequency': 15, u'level': u'Other'}]]}]}
```


25.3.9 Frame classification_metrics

classification_metrics (*self*, *label_column*, *pred_column*, *pos_label=None*, *beta=None*)

Model statistics of accuracy, precision, and others.

Parameters *label_column* : unicode

The name of the column containing the correct label for each instance.

pred_column : unicode

The name of the column containing the predicted label for each instance.

pos_label : None (default=None)

beta : float64 (default=None)

This is the beta value to use for F_β measure (default F1 measure is computed); must be greater than zero. Defaults is 1.

Returns : dict

object <object>.accuracy : double <object>.confusion_matrix : table
<object>.f_measure : double <object>.precision : double <object>.recall : double

Calculate the accuracy, precision, confusion_matrix, recall and F_β measure for a classification model.

- The **f_measure** result is the F_β measure for a classification model. The F_β measure of a binary classification model is the harmonic mean of precision and recall. If we let:

–beta $\equiv \beta$,

– T_P denotes the number of true positives,

– F_P denotes the number of false positives, and

– F_N denotes the number of false negatives

then:

$$F_\beta = (1 + \beta^2) * \frac{\frac{T_P}{T_P + F_P} * \frac{T_P}{T_P + F_N}}{\beta^2 * \frac{T_P}{T_P + F_P} + \frac{T_P}{T_P + F_N}}$$

The F_β measure for a multi-class classification model is computed as the weighted average of the F_β measure for each label, where the weight is the number of instances of each label. The determination of binary vs. multi-class is automatically inferred from the data.

- The **recall** result of a binary classification model is the proportion of positive instances that are correctly identified. If we let T_P denote the number of true positives and F_N denote the number of false negatives, then the model recall is given by $\frac{T_P}{T_P + F_N}$.

For multi-class classification models, the recall measure is computed as the weighted average of the recall for each label, where the weight is the number of instances of each label. The determination of binary vs. multi-class is automatically inferred from the data.

- The **precision** of a binary classification model is the proportion of predicted positive instances that are correctly identified. If we let T_P denote the number of true positives and F_P denote the number of false positives, then the model precision is given by: $\frac{T_P}{T_P + F_P}$.

For multi-class classification models, the precision measure is computed as the weighted average of the precision for each label, where the weight is the number of instances of each label. The determination of binary vs. multi-class is automatically inferred from the data.

- The **accuracy** of a classification model is the proportion of predictions that are correctly identified. If we let T_P denote the number of true positives, T_N denote the number of true negatives, and K denote the total number of classified instances, then the model accuracy is given by: $\frac{T_P + T_N}{K}$.

This measure applies to binary and multi-class classifiers.

- The **confusion_matrix** result is a confusion matrix for a binary classifier model, formatted for human readability.

Notes

The **confusion_matrix** is not yet implemented for multi-class classifiers.

25.3.10 *Frame* column_median

column_median (*self*, *data_column*, *weights_column=None*)

Calculate the (weighted) median of a column.

Parameters *data_column* : unicode

The column whose median is to be calculated.

weights_column : unicode (default=None)

The column that provides weights (frequencies) for the median calculation. Must contain numerical data. Default is all items have a weight of 1.

Returns : dict

varies The median of the values. If a weight column is provided and no weights are finite numbers greater than 0, None is returned. The type of the median returned is the same as the contents of the data column, so a column of Longs will result in a Long median and a column of Floats will result in a Float median.

The median is the least value X in the range of the distribution so that the cumulative weight of values strictly below X is strictly less than half of the total weight and the cumulative weight of values up to and including X is greater than or equal to one-half of the total weight.

All data elements of weight less than or equal to 0 are excluded from the calculation, as are all data elements whose weight is NaN or infinite. If a weight column is provided and no weights are finite numbers greater than 0, None is returned.

25.3.11 *Frame* column_mode

column_mode (*self*, *data_column*, *weights_column=None*, *max_modes_returned=None*)

Evaluate the weights assigned to rows.

Parameters *data_column* : unicode

Name of the column supplying the data.

weights_column : unicode (default=None)

Name of the column supplying the weights. Default is all items have weight of 1.

max_modes_returned : int32 (default=None)

Maximum number of modes returned. Default is 1.

Returns : dict

dict Dictionary containing summary statistics. The data returned is composed of multiple components:

mode [A mode is a data element of maximum net weight.] A set of modes is returned. The empty set is returned when the sum of the weights is 0. If the number of modes is less than or equal to the parameter `max_modes_returned`, then all modes of the data are returned. If the number of modes is greater than the `max_modes_returned` parameter, only the first `max_modes_returned` many modes (per a canonical ordering) are returned.

weight_of_mode [Weight of a mode.] If there are no data elements of finite weight greater than 0, the weight of the mode is 0. If no weights column is given, this is the number of appearances of each mode.

total_weight [Sum of all weights in the weight column.] This is the row count if no weights are given. If no weights column is given, this is the number of rows in the table with non-zero weight.

mode_count [The number of distinct modes in the data.] In the case that the data is very multimodal, this number may exceed `max_modes_returned`.

Calculate the modes of a column. A mode is a data element of maximum weight. All data elements of weight less than or equal to 0 are excluded from the calculation, as are all data elements whose weight is NaN or infinite. If there are no data elements of finite weight greater than 0, no mode is returned.

Because data distributions often have multiple modes, it is possible for a set of modes to be returned. By default, only one is returned, but by setting the optional parameter `max_modes_returned`, a larger number of modes can be returned.

25.3.12 *Frame* column_names

column_names

Column identifications in the current frame.

Parameters

Returns : list

list of names of all the frame's columns

Given a Frame object, *my_frame* accessing a frame. To get the column names:

```
>>> my_columns = my_frame.column_names
>>> print my_columns
```

Now, given there are three columns *col1*, *col2*, and *col3*, the result is:

```
["col1", "col2", "col3"]
```

25.3.13 *Frame* column_summary_statistics

column_summary_statistics (*self*, *data_column*, *weights_column=None*,
use_population_variance=None)

Calculate multiple statistics for a column.

Parameters **data_column** : unicode

The column to be statistically summarized. Must contain numerical data; all NaNs and infinite values are excluded from the calculation.

weights_column : unicode (default=None)

Name of column holding weights of column values.

use_population_variance : bool (default=None)

If true, the variance is calculated as the population variance. If false, the variance calculated as the sample variance. Because this option affects the variance, it affects the standard deviation and the confidence intervals as well. Default is false.

Returns : dict

dict Dictionary containing summary statistics. The data returned is composed of multiple components:

mean [[double | None]] Arithmetic mean of the data.

geometric_mean [[double | None]] Geometric mean of the data. None when there is a data element ≤ 0 , 1.0 when there are no data elements.

variance [[double | None]] None when there are ≤ 1 many data elements. Sample variance is the weighted sum of the squared distance of each data element from the weighted mean, divided by the total weight minus 1. None when the sum of the weights is ≤ 1 . Population variance is the weighted sum of the squared distance of each data element from the weighted mean, divided by the total weight.

standard_deviation [[double | None]] The square root of the variance. None when sample variance is being used and the sum of weights is ≤ 1 .

total_weight [long] The count of all data elements that are finite numbers. (In other words, after excluding NaNs and infinite values.)

minimum [[double | None]] Minimum value in the data. None when there are no data elements.

maximum [[double | None]] Maximum value in the data. None when there are no data elements.

mean_confidence_lower [[double | None]] Lower limit of the 95% confidence interval about the mean. Assumes a Gaussian distribution. None when there are no elements of positive weight.

mean_confidence_upper [[double | None]] Upper limit of the 95% confidence interval about the mean. Assumes a Gaussian distribution. None when there are no elements of positive weight.

bad_row_count [[double | None]] The number of rows containing a NaN or infinite value in either the data or weights column.

good_row_count [[double | None]] The number of rows not containing a NaN or infinite value in either the data or weights column.

positive_weight_count [[double | None]] The number of valid data elements with weight > 0. This is the number of entries used in the statistical calculation.

non_positive_weight_count [[double | None]] The number valid data elements with finite weight <= 0.

Notes

Sample Variance Sample Variance is computed by the following formula:

$$\left(\frac{1}{W-1} \right) * \sum_i (x_i - M)^2$$

where W is sum of weights over valid elements of positive weight, and M is the weighted mean.

Population Variance Population Variance is computed by the following formula:

$$\left(\frac{1}{W} \right) * \sum_i (x_i - M)^2$$

where W is sum of weights over valid elements of positive weight, and M is the weighted mean.

Standard Deviation The square root of the variance.

Logging Invalid Data A row is bad when it contains a NaN or infinite value in either its data or weights column. In this case, it contributes to bad_row_count; otherwise it contributes to good row count.

A good row can be skipped because the value in its weight column is less than or equal to 0. In this case, it contributes to non_positive_weight_count, otherwise (when the weight is greater than 0) it contributes to valid_data_weight_pair_count.

Equations `bad_row_count + good_row_count = # rows in the frame`

`positive_weight_count + non_positive_weight_count = good_row_count`

In particular, when no weights column is provided and all weights are 1.0,

`non_positive_weight_count = 0` and `positive_weight_count = good_row_count`

25.3.14 *Frame compute_misplaced_score*

compute_misplaced_score (*self*, *gravity*)

Parameters *gravity* : float64

Similarity measure for computing tension between 2 connected items

Returns : <bound method AtkEntityType.__name__ of

<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

25.3.15 *Frame copy*

copy (*self*, *columns=None*, *where=None*, *name=None*)

Create new frame from current frame.

Parameters *columns* : str | list of str | dict (default=None)

If not None, the copy will only include the columns specified. If dict, the string pairs represent a column renaming, {source_column_name: destination_column_name}

where : function (default=None)

If not None, only those rows for which the UDF evaluates to True will be copied.

name : str (default=None)

Name of the copied frame

Returns : Frame

A new Frame of the copied data.

Copy frame or certain frame columns entirely or filtered. Useful for frame query.

Examples

Build a Frame from a csv file with 5 million rows of data; call the frame “cust”:

```
>>> my_frame = ta.Frame(source="my_data.csv")
>>> my_frame.name("cust")
```

Given the frame has columns *id*, *name*, *hair*, and *shoe*. Copy it to a new frame:

```
>>> your_frame = my_frame.copy()
```

Now we have two frames of data, each with 5 million rows. Checking the names:

```
>>> print my_frame.name()
>>> print your_frame.name()
```

Gives the results:

```
"cust"
"frame_75401b7435d7132f5470ba35..."
```

Now, let’s copy *some* of the columns from the original frame:

```
>>> our_frame = my_frame.copy(['id', 'hair'])
```

Our new frame now has two columns, *id* and *hair*, and has 5 million rows. Let’s try that again, but this time change the name of the *hair* column to *color*:

```
>>> last_frame = my_frame.copy({'id': 'id', 'hair': 'color'})
```

25.3.16 Frame correlation

correlation (*self*, *data_column_names*)

Calculate correlation for two columns of current frame.

Parameters *data_column_names* : list

The names of 2 columns from which to compute the correlation.

Returns : dict

Pearson correlation coefficient of the two columns.

This method applies only to columns containing numerical data.

25.3.17 *Frame correlation_matrix*

correlation_matrix (*self*, *data_column_names*, *matrix_name=None*)

Calculate correlation matrix for two or more columns.

Parameters *data_column_names* : list

The names of the columns from which to compute the matrix.

matrix_name : unicode (default=None)

The name for the returned matrix Frame.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A Frame with the matrix of the correlation values for the columns.

This method applies only to columns containing numerical data.

25.3.18 *Frame count*

count (*self*, *where*)

Counts the number of rows which meet given criteria.

Parameters *where* : function

UDF which evaluates a row to a boolean

Returns : int

number of rows for which the where UDF evaluated to True.

25.3.19 *Frame covariance*

covariance (*self*, *data_column_names*)

Calculate covariance for exactly two columns.

Parameters *data_column_names* : list

The names of two columns from which to compute the covariance.

Returns : dict

Covariance of the two columns.

This method applies only to columns containing numerical data.

25.3.20 *Frame covariance_matrix*

covariance_matrix (*self*, *data_column_names*, *matrix_name=None*)

Calculate covariance matrix for two or more columns.

Parameters *data_column_names* : list

The names of the column from which to compute the matrix. Names should refer to a single column of type vector, or two or more columns of numeric scalars.

matrix_name : unicode (default=None)

The name of the new matrix.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A matrix with the covariance values for the columns.

This function applies only to columns containing numerical data.

25.3.21 *Frame cumulative_percent*

cumulative_percent (*self*, *sample_col*)

[BETA] Add column to frame with cumulative percent sum.

Parameters *sample_col* : unicode

The name of the column from which to compute the cumulative percent sum.

Returns : `_Unit`

A cumulative percent sum is computed by sequentially stepping through the rows, observing the column values and keeping track of the current percentage of the total sum accounted for at the current value.

Notes

This method applies only to columns containing numerical data. Although this method will execute for columns containing negative values, the interpretation of the result will change (for example, negative percentages).

25.3.22 *Frame cumulative_sum*

cumulative_sum (*self*, *sample_col*)

[BETA] Add column to frame with cumulative percent sum.

Parameters *sample_col* : unicode

The name of the column from which to compute the cumulative sum.

Returns : `_Unit`

A cumulative sum is computed by sequentially stepping through the rows, observing the column values and keeping track of the cumulative sum for each value.

Notes

This method applies only to columns containing numerical data.

25.3.23 *Frame dot_product*

dot_product (*self*, *left_column_names*, *right_column_names*, *dot_product_column_name*, *default_left_values=None*, *default_right_values=None*)
[ALPHA] Calculate dot product for each row in current frame.

Parameters *left_column_names* : list

Names of columns used to create the left vector (A) for each row. Names should refer to a single column of type vector, or two or more columns of numeric scalars.

right_column_names : list

Names of columns used to create right vector (B) for each row. Names should refer to a single column of type vector, or two or more columns of numeric scalars.

dot_product_column_name : unicode

Name of column used to store the dot product.

default_left_values : list (default=None)

Default values used to substitute null values in left vector. Default is None.

default_right_values : list (default=None)

Default values used to substitute null values in right vector. Default is None.

Returns : _Unit

Calculate the dot product for each row in a frame using values from two equal-length sequences of columns.

Dot product is computed by the following formula:

The dot product of two vectors $A = [a_1, a_2, \dots, a_n]$ and $B = [b_1, b_2, \dots, b_n]$ is $a_1 * b_1 + a_2 * b_2 + \dots + a_n * b_n$. The dot product for each row is stored in a new column in the existing frame.

Notes

If *default_left_values* or *default_right_values* are not specified, any null values will be replaced by zeros.

25.3.24 *Frame download*

download (*self*, *n=100*, *offset=0*, *columns=None*)
Download a frame from the server into client workspace.

Parameters *n* : int (default=100)

The number of rows to download to the client

offset : int (default=0)

The number of rows to skip before copying

columns : list (default=None)

Column filter, the names of columns to be included (default is all columns)

Returns : pandas.DataFrame

A new pandas dataframe object containing the downloaded frame data

Copies an trustedanalytics Frame into a Pandas DataFrame.

Examples

Frame *my_frame* accesses a frame with millions of rows of data. Get a sample of 500 rows:

```
>>> pandas_frame = my_frame.download( 500 )
```

We now have a new frame accessed by a pandas DataFrame *pandas_frame* with a copy of the first 500 rows of the original frame.

If we use the method with an offset like:

```
>>> pandas_frame = my_frame.take( 500, 100 )
```

We end up with a new frame accessed by the pandas DataFrame *pandas_frame* again, but this time it has a copy of rows 101 to 600 of the original frame.

25.3.25 Frame drop_columns

drop_columns (*self*, *columns*)

Remove columns from the frame.

Parameters **columns** : list

Column name OR list of column names to be removed from the frame.

Returns : _Unit

The data from the columns is lost.

Notes

It is not possible to delete all columns from a frame. At least one column needs to remain. If it is necessary to delete all columns, then delete the frame.

25.3.26 Frame drop_duplicates

drop_duplicates (*self*, *unique_columns=None*)

Modify the current frame, removing duplicate rows.

Parameters **unique_columns** : None (default=None)

Returns : _Unit

Remove data rows which are the same as other rows. The entire row can be checked for duplication, or the search for duplicates can be limited to one or more columns. This modifies the current frame.

25.3.27 *Frame drop_rows*

drop_rows (*self*, *predicate*)

Erase any row in the current frame which qualifies.

Parameters **predicate** : function

UDF which evaluates a row to a boolean; rows that answer True are dropped from the Frame

Examples

For this example, `my_frame` is a Frame object accessing a frame with lots of data for the attributes of lions, tigers, and ligers. Get rid of the lions and tigers:

```
>>> my_frame.drop_rows(lambda row: row.animal_type == "lion" or
...                     row.animal_type == "tiger")
```

Now the frame only has information about ligers.

More information on a UDF can be found at [Python User Functions](#).

25.3.28 *Frame ecdf*

ecdf (*self*, *column*, *result_frame_name=None*)

Builds new frame with columns for data and distribution.

Parameters **column** : unicode

The name of the input column containing sample.

result_frame_name : unicode (default=None)

A name for the resulting frame which is created by this operation.

Returns : <bound method `AtkEntityType.__name__` of

<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A new Frame containing each distinct value in the sample and its corresponding ECDF value.

Generates the *empirical cumulative distribution* for the input column.

25.3.29 *Frame entropy*

entropy (*self*, *data_column*, *weights_column=None*)

Calculate the Shannon entropy of a column.

Parameters **data_column** : unicode

The column whose entropy is to be calculated.

weights_column : unicode (default=None)

The column that provides weights (frequencies) for the entropy calculation. Must contain numerical data. Default is using uniform weights of 1 for all items.

Returns : dict

Entropy.

The data column is weighted via the weights column. All data elements of weight ≤ 0 are excluded from the calculation, as are all data elements whose weight is NaN or infinite. If there are no data elements with a finite weight greater than 0, the entropy is zero.

25.3.30 *Frame export_to_csv*

export_to_csv (*self*, *folder_name*, *separator=None*, *count=None*, *offset=None*)

Write current frame to HDFS in csv format.

Parameters **folder_name** : unicode

The HDFS folder path where the files will be created.

separator : None (default=None)

count : int32 (default=None)

The number of records you want. Default, or a non-positive value, is the whole frame.

offset : int32 (default=None)

The number of rows to skip before exporting to the file. Default is zero (0).

Returns : _Unit

Export the frame to a file in csv format as a Hadoop file.

25.3.31 *Frame export_to_hbase*

export_to_hbase (*self*, *table_name*, *key_column_name=None*, *family_name=None*)

Write current frame to HBase table.

Parameters **table_name** : unicode

The name of the HBase table that will contain the exported frame

key_column_name : unicode (default=None)

The name of the column to be used as row key in hbase table

family_name : unicode (default=None)

The family name of the HBase table that will contain the exported frame

Returns : _Unit

Table must exist in HBase. Export of Vectors is not currently supported.

25.3.32 *Frame export_to_hive*

export_to_hive (*self*, *table_name*)

Write current frame to Hive table.

Parameters *table_name* : unicode

The name of the Hive table that will contain the exported frame

Returns : `_Unit`

Table must not exist in Hive. Export of Vectors is not currently supported.

25.3.33 *Frame export_to_jdbc*

export_to_jdbc (*self*, *table_name*, *connector_type=None*, *url=None*, *driver_name=None*, *query=None*)

Write current frame to Jdbc table.

Parameters *table_name* : unicode

jdbc table name

connector_type : unicode (default=None)

(optional) jdbc connector type

url : unicode (default=None)

(optional) connection url (includes server name, database name, user acct and password)

driver_name : unicode (default=None)

(optional) driver name

query : unicode (default=None)

(optional) query for filtering. Not supported yet.

Returns : `_Unit`

Table will be created or appended to. Export of Vectors is not currently supported.

25.3.34 *Frame export_to_json*

export_to_json (*self*, *folder_name*, *count=None*, *offset=None*)

Write current frame to HDFS in JSON format.

Parameters *folder_name* : unicode

The HDFS folder path where the files will be created.

count : int32 (default=None)

The number of records you want. Default, or a non-positive value, is the whole frame.

offset : int32 (default=None)

The number of rows to skip before exporting to the file. Default is zero (0).

Returns : `_Unit`

Export the frame to a file in JSON format as a Hadoop file.

25.3.35 *Frame* filter

filter (*self*, *predicate*)

Select all rows which satisfy a predicate.

Parameters **predicate** : function

UDF which evaluates a row to a boolean; rows that answer False are dropped from the Frame

Modifies the current frame to save defined rows and delete everything else.

Examples

For this example, *my_frame* is a Frame object with lots of data for the attributes of lizards, frogs, and snakes. Get rid of everything, except information about lizards and frogs:

```
>>> def my_filter(row):  
...     return row['animal_type'] == 'lizard' or  
...     row['animal_type'] == "frog"  
  
>>> my_frame.filter(my_filter)
```

The frame now only has data about lizards and frogs.

More information on a UDF can be found at [Python User Functions](#).

25.3.36 *Frame* flatten_column

flatten_column (*self*, *column*, *delimiter=None*)

Spread data to multiple rows based on cell data.

Parameters **column** : unicode

The column to be flattened.

delimiter : unicode (default=None)

The delimiter string. Default is comma (,).

Returns : `_Unit`

Splits cells in the specified column into multiple rows according to a string delimiter. New rows are a full copy of the original row, but the specified column only contains one value. The original row is deleted.

25.3.37 *Frame* `get_error_frame`

get_error_frame (*self*)

Get a frame with error recordings.

Parameters

When a frame is created, another frame is transparently created to capture parse errors.

Returns **Frame** : error frame object

A new object accessing a frame that contains the parse errors of the currently active Frame or None if no error frame exists.

25.3.38 *Frame* `group_by`

group_by (*self*, *group_by_columns*, *aggregation_arguments=None*)

[BETA] Create summarized frame.

Parameters **group_by_columns** : list

Column name or list of column names

aggregation_arguments : dict (default=None)

Aggregation function based on entire row, and/or dictionaries (one or more) of { column name str : aggregation function(s) }.

Returns : Frame

A new frame with the results of the `group_by`

Creates a new frame and returns a Frame object to access it. Takes a column or group of columns, finds the unique combination of values, and creates unique rows with these column values. The other columns are combined according to the aggregation argument(s).

Notes

- Column order is not guaranteed when columns are added
- The column names created by aggregation functions in the new frame are the original column name appended with the ‘_’ character and the aggregation function. For example, if the original field is *a* and the function is *avg*, the resultant column is named *a_avg*.
- An aggregation argument of *count* results in a column named *count*.
- The aggregation function *agg.count* is the only full row aggregation function supported at this time.
- Aggregation currently supports using the following functions:
 - avg
 - count
 - count_distinct
 - max
 - min

- stdev
- sum
- var (see glossary *Bias vs Variance*)

Examples

For setup, we will use a Frame *my_frame* accessing a frame with a column *a*:

```
>>> my_frame.inspect()

a:str
/-----/
cat
apple
bat
cat
bat
cat
```

Create a new frame, combining similar values of column *a*, and count how many of each value is in the original frame:

```
>>> new_frame = my_frame.group_by('a', agg.count)
>>> new_frame.inspect()

a:str      count:int
/-----/
cat         3
apple       1
bat         2
```

In this example, ‘*my_frame*’ is accessing a frame with three columns, *a*, *b*, and *c*:

```
>>> my_frame.inspect()

a:int  b:str  c:float
/-----/
1      alpha  3.0
1      bravo  5.0
1      alpha  5.0
2      bravo  8.0
2      bravo 12.0
```

Create a new frame from this data, grouping the rows by unique combinations of column *a* and *b*. Average the value in *c* for each group:

```
>>> new_frame = my_frame.group_by(['a', 'b'], {'c' : agg.avg})
>>> new_frame.inspect()

a:int  b:str  c_avg:float
/-----/
1      alpha  4.0
1      bravo  5.0
2      bravo 10.0
```

For this example, we use *my_frame* with columns *a*, *c*, *d*, and *e*:


```
>>> my_frame.inspect()

  a:str   c:int   d:float e:int
/-----/
ape      1      4.0     9
ape      1      8.0     8
big      1      5.0     7
big      1      6.0     6
big      1      8.0     5
```

Create a new frame from this data, grouping the rows by unique combinations of column *a* and *c*. Count each group; for column *d* calculate the average, sum and minimum value. For column *e*, save the maximum value:

```
>>> new_frame = my_frame.group_by(['a', 'c'], agg.count,
... {'d': [agg.avg, agg.sum, agg.min], 'e': agg.max})

  a    c   count  d_avg  d_sum  d_min  e_max
str  int  int    float  float  float  int
/-----/
ape  1    2     6.0   12.0   4.0    9
big  1    3     6.333 19.0   5.0    7
```

For further examples, see [Group by \(and aggregate\)](#)..

25.3.39 Frame histogram

histogram (*self*, *column_name*, *num_bins*=None, *weight_column_name*=None, *bin_type*='equalwidth')
[BETA] Compute the histogram for a column in a frame.

Parameters **column_name** : unicode

Name of column to be evaluated.

num_bins : int32 (default=None)

Number of bins in histogram. Default is Square-root choice will be used (in other words $\text{math.floor}(\text{math.sqrt}(\text{frame.row_count}))$).

weight_column_name : unicode (default=None)

Name of column containing weights. Default is all observations are weighted equally.

bin_type : unicode (default=equalwidth)

The type of binning algorithm to use: ["equalwidth"|"equaldepth"] Defaults is "equalwidth".

Returns : dict

histogram A Histogram object containing the result set. The data returned is composed of multiple components:

cutoffs [array of float] A list containing the edges of each bin.

hist [array of float] A list containing count of the weighted observations found in each bin.

density [array of float] A list containing a decimal containing the percentage of observations found in the total set per bin.

Compute the histogram of the data in a column. The returned value is a Histogram object containing 3 lists one each for: the cutoff points of the bins, size of each bin, and density of each bin.

Notes

The `num_bins` parameter is considered to be the maximum permissible number of bins because the data may dictate fewer bins. With equal depth binning, for example, if the column to be binned has 10 elements with only 2 distinct values and the `num_bins` parameter is greater than 2, then the number of actual number of bins will only be 2. This is due to a restriction that elements with an identical value must belong to the same bin.

25.3.40 *Frame inspect*

inspect (*self*, *n=10*, *offset=0*, *columns=None*, *wrap=None*, *truncate=None*, *round=None*, *width=80*, *margin=None*)

Prints the frame data in readable format.

Parameters **n** : int (default=10)

The number of rows to print.

offset : int (default=0)

The number of rows to skip before printing.

columns : int (default=None)

Filter columns to be included. By default, all columns are included

wrap : int or 'stripes' (default=None)

If set to 'stripes' then inspect prints rows in stripes; if set to an integer N, rows will be printed in clumps of N columns, where the columns are wrapped

truncate : int (default=None)

If set to integer N, all strings will be truncated to length N, including a tagged ellipses

round : int (default=None)

If set to integer N, all floating point numbers will be rounded and truncated to N digits

width : int (default=80)

If set to integer N, the print out will try to honor a max line width of N

margin : int (default=None)

('stripes' mode only) If set to integer N, the margin for printing names in a stripe will be limited to N characters

Examples

Given a frame of data and a Frame to access it. To look at the first 4 rows of data:

```
>>> print my_frame.inspect(4)

column defs ->  animal:str  name:str   age:int   weight:float
               /-----/
frame data ->   human      George    8         542.5
               human      Ursula     6         495.0
```

ape	Ape	41	400.0
elephant	Shep	5	8630.0

For other examples, see *Inspect The Data*.

25.3.41 *Frame join*

join (*self*, *right*, *left_on*, *right_on*=None, *how*=‘inner’, *name*=None)
 [BETA] Join operation on one or two frames, creating a new frame.

Parameters **right** : Frame

Another frame to join with

left_on : str

Name of the column in the left frame used to match up the two frames.

right_on : str (default=None)

Name of the column in the right frame used to match up the two frames. Default is the same as the left frame.

how : str (default=inner)

How to qualify the data to be joined together. Must be one of the following: ‘left’, ‘right’, ‘inner’, ‘outer’. Default is ‘inner’

name : str (default=None)

Name of the result grouped frame

Returns : Frame

A new frame with the results of the join

Create a new frame from a SQL JOIN operation with another frame. The frame on the ‘left’ is the currently active frame. The frame on the ‘right’ is another frame. This method takes a column in the left frame and matches its values with a column in the right frame. Using the default ‘how’ option [‘inner’] will only allow data in the resultant frame if both the left and right frames have the same value in the matching column. Using the ‘left’ ‘how’ option will allow any data in the resultant frame if it exists in the left frame, but will allow any data from the right frame if it has a value in its column which matches the value in the left frame column. Using the ‘right’ option works similarly, except it keeps all the data from the right frame and only the data from the left frame when it matches. The ‘outer’ option provides a frame with data from both frames where the left and right frames did not have the same value in the matching column.

Notes

When a column is named the same in both frames, it will result in two columns in the new frame. The column from the *left* frame (originally the current frame) will be copied and the column name will have the string “_L” added to it. The same thing will happen with the column from the *right* frame, except its name has the string “_R” appended. The order of columns after this method is called is not guaranteed.

It is recommended that you rename the columns to meaningful terms prior to using the `join` method. Keep in mind that unicode in column names will likely cause the `drop_frames()` method (and others) to fail!

Examples

For this example, we will use a Frame *my_frame* accessing a frame with columns *a*, *b*, *c*, and a Frame *your_frame* accessing a frame with columns *a*, *d*, *e*. Join the two frames keeping only those rows having the same value in column *a*:

```
>>> print my_frame.inspect()

  a:unicode  b:unicode  c:unicode
/-----/
alligator   bear       cat
apple       berry      cantaloupe
auto        bus        car
mirror      frog       ball

>>> print your_frame.inspect()

  b:unicode  c:int  d:unicode
/-----/
berry       5218  frog
blue        0    log
bus         871  dog

>>> joined_frame = my_frame.join(your_frame, 'b', how='inner')
```

Now, *joined_frame* is a Frame accessing a frame with the columns *a*, *b*, *c_L*, *c_R*, and *d*. The data in the new frame will be from the rows where column 'a' was the same in both frames.

```
>>> print joined_frame.inspect()

  a:unicode  b:unicode  c_L:unicode  c_R:int64  d:unicode
/-----/
apple       berry      cantaloupe    5218      frog
auto        bus        car              871      dog
```

More examples can be found in the [user manual](#).

25.3.42 Frame label_propagation

label_propagation(*self*, *src_col_name*, *dest_col_name*, *weight_col_name*, *src_label_col_name*, *result_col_name=None*, *max_iterations=None*, *convergence_threshold=None*, *alpha=None*)

Label Propagation on Gaussian Random Fields.

Parameters *src_col_name* : unicode

The column name for the source vertex id.

dest_col_name : unicode

The column name for the destination vertex id.

weight_col_name : unicode

The column name for the edge weight.

src_label_col_name : unicode

The column name for the label properties for the source vertex.

result_col_name : unicode (default=None)

The column name for the results (holding the post labels for the vertices).

max_iterations : int32 (default=None)

The maximum number of supersteps that the algorithm will execute. The valid value range is all positive int. Default is 10.

convergence_threshold : float32 (default=None)

The amount of change in cost function that will be tolerated at convergence. If the change is less than this threshold, the algorithm exits earlier before it reaches the maximum number of supersteps. The valid value range is all float and zero. Default is 0.00000001f.

alpha : float32 (default=None)

The tradeoff parameter that controls how much influence an external classifier's prediction contributes to the final prediction. This is for the case where an external classifier is available that can produce initial probabilistic classification on unlabeled examples, and the option allows incorporating external classifier's prediction into the LP training process. The valid value range is [0.0,1.0]. Default is 0.

Returns : dict

A 2-column frame:

vertex: int A vertex id.

result [Vector (long)] label vector for the results (for the node id in column 1)

Label Propagation on Gaussian Random Fields.

This algorithm is presented in X. Zhu and Z. Ghahramani. *Learning from labeled and unlabeled data with label propagation*. Technical Report CMU-CALD-02-107, CMU, 2002¹.

Label Propagation (LP)

LP (Label Propagation) is a message passing technique for inputting or *smoothing* labels in partially-labelled datasets. Labels are propagated from *labeled* data to *unlabeled* data along a graph encoding similarity relationships among data points. The labels of known data can be probabilistic, in other words, a known point can be represented with fuzzy labels such as 90% label 0 and 10% label 1. The inverse distance between data points is represented by edge weights, with closer points having a higher weight (stronger influence on posterior estimates) than points farther away. LP has been used for many problems, particularly those involving a similarity measure between data points. Our implementation is based on Zhu and Ghahramani's 2002 paper, *Learning from labeled and unlabeled data*.²

The Label Propagation Algorithm

In LP, all nodes start with a prior distribution of states and the initial messages vertices pass to their neighbors are simply their prior beliefs. If certain observations have states that are known deterministically, they can be given a prior probability of 100% for their true state and 0% for all others. Unknown observations should be given uninformative priors.

Each node, i , receives messages from its k neighbors and updates its beliefs by taking a weighted average of its current beliefs and a weighted average of the messages received from its neighbors.

¹<http://www.cs.cmu.edu/~zhuxj/pub/CMU-CALD-02-107.pdf>

²<http://www.cs.cmu.edu/~zhuxj/pub/CMU-CALD-02-107.pdf>

The updated beliefs for node i are:

$$updated\ beliefs_i = \lambda * (prior\ belief_i) + (1 - \lambda) * \sum_k w_{i,k} * previous\ belief_k$$

where $w_{i,k}$ is the normalized weight between nodes i and k , normalized such that the sum of all weights to neighbors is 1.

λ is a leaning parameter. If λ is greater than zero, updated probabilities will be anchored in the direction of prior beliefs.

The final distribution of state probabilities will also tend to be biased in the direction of the distribution of initial beliefs. For the first iteration of updates, nodes' previous beliefs are equal to the priors, and, in each future iteration, previous beliefs are equal to their beliefs as of the last iteration. All beliefs for every node will be updated in this fashion, including known observations, unless `anchor_threshold` is set. The `anchor_threshold` parameter specifies a probability threshold above which beliefs should no longer be updated. Hence, with an `anchor_threshold` of 0.99, observations with states known with 100% certainty will not be updated by this algorithm.

This process of updating and message passing continues until the convergence criteria is met, or the maximum number of *supersteps* is reached. A node is said to converge if the total change in its cost function is below the convergence threshold. The cost function for a node is given by:

$$cost = \sum_k w_{i,k} * \left[(1 - \lambda) * [previous\ belief_i^2 - w_{i,k} * previous\ belief_i * previous\ belief_k] + 0.5 * \lambda * (previous\ belief_i - prior_i)^2 \right]$$

Convergence is a local phenomenon; not all nodes will converge at the same time. It is also possible that some (most) nodes will converge and others will not converge. The algorithm requires all nodes to converge before declaring global convergence. If this condition is not met, the algorithm will continue up to the maximum number of *supersteps*.

25.3.43 Frame loadhbase

loadhbase (*self*, *table_name*, *schema*, *start_tag*=None, *end_tag*=None)

Append data from an hBase table into an existing (possibly empty) FrameRDD

Parameters *table_name* : unicode

hbase table name

schema : list

hbase schema as a list of tuples (columnFamily, columnName, dataType for cell value)

start_tag : unicode (default=None)

optional start tag for filtering

end_tag : unicode (default=None)

optional end tag for filtering

Returns : <bound method `AtkEntityType.__name__` of

<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

the initial FrameRDD with the hbase data appended

Append data from an hBase table into an existing (possibly empty) FrameRDD

25.3.44 *Frame loadhive*

loadhive (*self, query*)

Append data from a hive table into an existing (possibly empty) frame

Parameters **query** : unicode

Initial query to run at load time

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
the initial frame with the hive data appended

Append data from a hive table into an existing (possibly empty) frame

25.3.45 *Frame loadjdbc*

loadjdbc (*self, table_name, connector_type=None, url=None, driver_name=None, query=None*)

Append data from a Jdbc table into an existing (possibly empty) frame

Parameters **table_name** : unicode

table name

connector_type : unicode (default=None)

(optional) connector type

url : unicode (default=None)

(optional) connection url (includes server name, database name, user acct and password)

driver_name : unicode (default=None)

(optional) driver name

query : unicode (default=None)

(optional) query for filtering. Not supported yet.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
the initial frame with the Jdbc data appended

Append data from a Jdbc table into an existing (possibly empty) frame

25.3.46 *Frame loopy_belief_propagation*

loopy_belief_propagation (*self, src_col_name, dest_col_name, weight_col_name, src_label_col_name, result_col_name=None, ignore_vertex_type=None, max_iterations=None, convergence_threshold=None, anchor_threshold=None, smoothing=None, max_product=None, power=None*)

Message passing to infer state probabilities.

Parameters `src_col_name` : unicode

The column name for the source vertex id.

`dest_col_name` : unicode

The column name for the destination vertex id.

`weight_col_name` : unicode

The column name for the edge weight.

`src_label_col_name` : unicode

The column name for the label properties for the source vertex.

`result_col_name` : unicode (default=None)

The column name for the results (holding the post labels for the vertices).

`ignore_vertex_type` : bool (default=None)

If True, all vertex will be treated as training data. Default is False.

`max_iterations` : int32 (default=None)

The maximum number of supersteps that the algorithm will execute. The valid value range is all positive int. The default value is 10.

`convergence_threshold` : float32 (default=None)

The amount of change in cost function that will be tolerated at convergence. If the change is less than this threshold, the algorithm exits earlier before it reaches the maximum number of supersteps. The valid value range is all float and zero. The default value is 0.00000001f.

`anchor_threshold` : float64 (default=None)

The parameter that determines if a node's posterior will be updated or not. If a node's maximum prior value is greater than this threshold, the node will be treated as anchor node, whose posterior will inherit from prior without update. This is for the case where we have confident prior estimation for some nodes and don't want the algorithm to update these nodes. The valid value range is in [0, 1]. Default is 1.0.

`smoothing` : float32 (default=None)

The Ising smoothing parameter. This parameter adjusts the relative strength of closeness encoded edge weights, similar to the width of Gaussian distribution. Larger value implies smoother decay and the edge weight becomes less important. Default is 2.0.

`max_product` : bool (default=None)

Should LBP (Loopy Belief Propagation) use max_product or not. Default is False.

`power` : float32 (default=None)

Power coefficient for power edge potential. Default is 0.

Returns : dict

a 2-column frame:

vertex: int A vertex id.

result [Vector (long)] label vector for the results (for the node id in column 1).

Loopy belief propagation on *Markov Random Fields* (MRF). *Belief Propagation* (BP) was originally designed for acyclic graphical models, then it was found that the BP (Belief Propagation) algorithm can be used in general graphs. The algorithm is then sometimes called “Loopy” Belief Propagation (LBP), because graphs typically contain cycles, or loops.

Loopy Belief Propagation (LBP)

Loopy Belief Propagation (LBP) is a message passing algorithm for inferring state probabilities, given a graph and a set of noisy initial estimates. The LBP implementation assumes that the joint distribution of the data is given by a Boltzmann distribution.

For more information about LBP, see: “K. Murphy, Y. Weiss, and M. Jordan, Loopy-belief Propagation for Approximate Inference: An Empirical Study, UAI 1999.”

LBP has a wide range of applications in structured prediction, such as low-level vision and influence spread in social networks, where we have prior noisy predictions for a large set of random variables and a graph encoding relationships between those variables.

The algorithm performs approximate inference on an *undirected graph* of hidden variables, where each variable is represented as a node, and each edge encodes relations to its neighbors. Initially, a prior noisy estimate of state probabilities is given to each node, then the algorithm infers the posterior distribution of each node by propagating and collecting messages to and from its neighbors and updating the beliefs.

In graphs containing loops, convergence is not guaranteed, though LBP has demonstrated empirical success in many areas and in practice often converges close to the true joint probability distribution.

Discrete Loopy Belief Propagation

LBP is typically considered a *semi-supervised machine learning* algorithm as

1. there is typically no ground truth observation of states
2. the algorithm is primarily concerned with estimating a joint probability function rather than with *classification* or point prediction.

The standard (discrete) LBP algorithm requires a set of probability thresholds to be considered a classifier. Nonetheless, the discrete LBP algorithm allows Test/Train/Validate splits of the data and the algorithm will treat “Train” observations differently from “Test” and “Validate” observations. Vertices labelled with “Test” or “Validate” will be treated as though they have uninformative (uniform) priors and are allowed to receive messages, but not send messages. This simulates a “scoring scenario” in which a new observation is added to a graph containing fully trained LBP posteriors, the new vertex is scored based on received messages, but the full LBP algorithm is not repeated in full. This behavior can be turned off by setting the `ignore_vertex_type` parameter to `True`. When `ignore_vertex_type=True`, all nodes will be considered “Train” regardless of their sample type designation. The Gaussian (continuous) version of LBP does not allow Train/Test/Validate splits.

The standard LBP algorithm included with the toolkit assumes an ordinal and cardinal set of discrete states. For notational convenience, we’ll denote the value of state s_i as i , and the prior probability of state s_i as $prior_i$.

Each node sends out initial messages of the form:

$$\ln \left(\sum_{s_j} \exp \left(-\frac{|i-j|^p}{n-1} * w * s + \ln(prior_i) \right) \right)$$

Where

- w is the weight between the messages destination and origin vertices
- s is the *smoothing* parameter
- p is the power parameter

• n is the number of states

The larger the weight between two nodes, or the higher the smoothing parameter, the more neighboring vertices are assumed to “agree” on states. We represent messages as sums of log probabilities rather than products of non-logged probabilities which makes it easier to subtract messages in the future steps of the algorithm. Also note that the states are cardinal in the sense that the “pull” of state i on state j depends on the distance between i and j . The *power* parameter intensifies the rate at which the pull of distant states drops off.

In order for the algorithm to work properly, all edges of the graph must be bidirectional. In other words, messages need to be able to flow in both directions across every edge. Bidirectional edges can be enforced during graph building, but the LBP function provides an option to do an initial check for bidirectionality using the `bidirectional_check=True` option. If not all the edges of the graph are bidirectional, the algorithm will return an error.

Look at a case where a node has two states, 0 and 1. The 0 state has a prior probability of 0.9 and the 1 state has a prior probability of 0.2. The states have uniform weights of 1, power of 1 and a smoothing parameter of 2. The nodes initial message would be $[\ln(0.2 + 0.8e^{-2}), \ln(0.8 + 0.2e^{-2})]$, which gets sent to each of that node’s neighbors. Note that messages will typically not be proper probability distributions, hence each message is normalized so that the probability of all states sum to 1 before being sent out. For simplicity of discussion, we will consider all messages as normalized messages.

After nodes have sent out their initial messages, they then update their beliefs based on messages that they have received from their neighbors, denoted by the set k .

Updated Posterior Beliefs:

$$\ln(\text{newbelief}) = \alpha \exp \left[\ln(\text{prior}) + \sum_k \text{message}_k \right]$$

Note that the messages in the above equation are still in log form. Nodes then send out new messages which take the same form as their initial messages, with updated beliefs in place of priors and subtracting out the information previously received from the new message’s recipient. The recipient’s prior message is subtracted out to prevent feedback loops of nodes “learning” from themselves.

$$\ln \left(\sum_{s_j} \exp \left(-\frac{|i-j|^p}{n-1} * w * s + \ln(\text{newbelief}_i) - \text{previous message from recipient} \right) \right)$$

In updating beliefs, new beliefs tend to be most influenced by the largest message. Setting the `max_product` option to “True” ignores all incoming messages other than the strongest signal. Doing this results in approximate solutions, but requires significantly less memory and run-time than the more exact computation. Users should consider this option when processing power is a constraint and approximate solutions to LBP will be sufficient.

This process of updating and message passing continues until the convergence criteria is met or the maximum number of *supersteps* is reached without converging. A node is said to converge if the total change in its distribution (the sum of absolute value changes in state probabilities) is less than the `convergence_threshold` parameter. Convergence is a local phenomenon; not all nodes will converge at the same time. It is also possible for some (most) nodes to converge and others to never converge. The algorithm requires all nodes to converge before declaring that the algorithm has converged overall. If this condition is not met, the algorithm will continue up to the maximum number of *supersteps*.

See: http://en.wikipedia.org/wiki/Belief_propagation.

25.3.47 Frame name

name

Set or get the name of the frame object.

Parameters

Change or retrieve frame object identification. Identification names must start with a letter and are limited to alphanumeric characters and the `_` character.

Examples

```
>>> my_frame.name

"csv_data"

>>> my_frame.name = "cleaned_data"
>>> my_frame.name

"cleaned_data"
```

25.3.48 *Frame* quantiles

quantiles (*self*, *column_name*, *quantiles*)

New frame with Quantiles and their values.

Parameters **column_name** : unicode

The column to calculate quantiles.

quantiles : list

What is being requested.

Returns : <bound method `AtkEntityType.__name__` of

<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A new frame with two columns (float64): requested Quantiles and their respective values.

Calculate quantiles on the given column.

25.3.49 *Frame* rename_columns

rename_columns (*self*, *names*)

Rename columns

Parameters **names** : None

Returns : `_Unit`

25.3.50 *Frame row_count*

row_count

Number of rows in the current frame.

Parameters

Returns : int

The number of rows in the frame

Get the number of rows:

```
>>> my_frame.row_count
```

The result given is:

```
81734
```

25.3.51 *Frame schema*

schema

Current frame column names and types.

Parameters

Returns : list

list of tuples of the form (<column name>, <data type>)

The schema of the current frame is a list of column names and associated data types. It is retrieved as a list of tuples. Each tuple has the name and data type of one of the frame's columns.

Examples

Given that we have an existing data frame *my_data*, create a Frame, then show the frame schema:

```
>>> BF = ta.get_frame('my_data')
>>> print BF.schema
```

The result is:

```
[("col1", str), ("col2", numpy.int32)]
```

25.3.52 *Frame sort*

sort (*self*, *columns*, *ascending=True*)

[BETA] Sort the data in a frame.

Parameters **columns** : str | list of str | list of tuples

Either a column name, a list of column names, or a list of tuples where each tuple is a name and an ascending bool value.

ascending : bool (default=True)

True for ascending, False for descending.

Sort a frame by column values either ascending or descending.

Examples

Sort a single column:

```
>>> frame.sort('column_name')
```

Sort a single column ascending:

```
>>> frame.sort('column_name', True)
```

Sort a single column descending:

```
>>> frame.sort('column_name', False)
```

Sort multiple columns:

```
>>> frame.sort(['col1', 'col2'])
```

Sort multiple columns ascending:

```
>>> frame.sort(['col1', 'col2'], True)
```

Sort multiple columns descending:

```
>>> frame.sort(['col1', 'col2'], False)
```

Sort multiple columns: 'col1' ascending and 'col2' descending:

```
>>> frame.sort([ ('col1', True), ('col2', False) ])
```

25.3.53 *Frame sorted_k*

sorted_k (*self, k, column_names_and_ascending, reduce_tree_depth=None*)

[ALPHA] Get a sorted subset of the data.

Parameters **k** : int32

Number of sorted records to return.

column_names_and_ascending : list

Column names to sort by, and true to sort column by ascending order, or false for descending order.

reduce_tree_depth : int32 (default=None)

Advanced tuning parameter which determines the depth of the reduce-tree for the sorted_k plugin. This plugin uses Spark's treeReduce() for scalability. The default depth is 2.

Returns : <bound method AtkEntityType.__name__ of

<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A new frame with the first *k* sorted rows from the original frame.

Take the first *k* (sorted) rows for the currently active Frame. Rows are sorted by column values in either ascending or descending order.

Returning the first *k* (sorted) rows is more efficient than sorting the entire frame when *k* is much smaller than the number of rows in the frame.

Notes

The number of sorted rows (*k*) should be much smaller than the number of rows in the original frame.

In particular:

1. The number of sorted rows (*k*) returned should fit in Spark driver memory.

The maximum size of serialized results that can fit in the Spark driver is set by the Spark configuration parameter *spark.driver.maxResultSize*.

2. If you encounter a Kryo buffer overflow exception, increase the Spark

configuration parameter *spark.kryoserializer.buffer.max.mb*.

3. Use `Frame.sort()` instead if the number of sorted rows (*k*) is

very large (i.e., cannot fit in Spark driver memory).

25.3.54 *Frame* status

status

Current frame life cycle status.

Parameters

Returns : str

Status of the frame

One of three statuses: Active, Deleted, Deleted_Final Active: Frame is available for use Deleted: Frame has been scheduled for deletion can be unscheduled by modifying Deleted_Final: Frame's backend files have been removed from disk.

Examples

Given that we have an existing data frame *my_data*, create a Frame, then show the frame schema:

```
>>> BF = ta.get_frame('my_data')
>>> print BF.status
```

The result is:

```
u'Active'
```

25.3.55 *Frame* take

take (*self*, *n*, *offset=0*, *columns=None*)

Get data subset.

Parameters *n* : int

The number of rows to copy to the client from the frame.

offset : int (default=0)

The number of rows to skip before starting to copy

columns : str | iterable of str (default=None)

If not None, only the given columns' data will be provided. By default, all columns are included

Returns : list

A list of lists, where each contained list is the data for one row.

Take a subset of the currently active Frame.

Notes

The data is considered 'unstructured', therefore taking a certain number of rows, the rows obtained may be different every time the command is executed, even if the parameters do not change.

Examples

Frame *my_frame* accesses a frame with millions of rows of data. Get a sample of 5000 rows:

```
>>> my_data_list = my_frame.take( 5000 )
```

We now have a list of data from the original frame.

```
>>> print my_data_list

[[ 1, "text", 3.1415962 ]
 [ 2, "bob", 25.0 ]
 [ 3, "weave", .001 ]
 ...]
```

If we use the method with an offset like:

```
>>> my_data_list = my_frame.take( 5000, 1000 )
```

We end up with a new list, but this time it has a copy of the data from rows 1001 to 5000 of the original frame.

25.3.56 *Frame* tally

tally (*self*, *sample_col*, *count_val*)

[BETA] Count number of times a value is seen.

Parameters `sample_col` : unicode

The name of the column from which to compute the cumulative count.

`count_val` : unicode

The column value to be used for the counts.

Returns : `_Unit`

A cumulative count is computed by sequentially stepping through the rows, observing the column values and keeping track of the the number of times the specified *count_value* has been seen.

25.3.57 *Frame tally_percent*

tally_percent (*self, sample_col, count_val*)

[BETA] Compute a cumulative percent count.

Parameters `sample_col` : unicode

The name of the column from which to compute the cumulative sum.

`count_val` : unicode

The column value to be used for the counts.

Returns : `_Unit`

A cumulative percent count is computed by sequentially stepping through the rows, observing the column values and keeping track of the percentage of the total number of times the specified *count_value* has been seen up to the current value.

25.3.58 *Frame top_k*

top_k (*self, column_name, k, weights_column=None*)

Most or least frequent column values.

Parameters `column_name` : unicode

The column whose top (or bottom) K distinct values are to be calculated.

`k` : int32

Number of entries to return (If k is negative, return bottom k).

weights_column : unicode (default=None)

The column that provides weights (frequencies) for the topK calculation. Must contain numerical data. Default is 1 for all items.

Returns : <bound method `AtkEntityType.__name__` of

<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

An object with access to the frame of data.

Calculate the top (or bottom) K distinct values by count of a column. The column can be weighted. All data elements of weight ≤ 0 are excluded from the calculation, as are all data elements whose weight is NaN or infinite. If there are no data elements of finite weight > 0 , then topK is empty.

25.3.59 *Frame* `unflatten_column`

unflatten_column (*self*, *composite_key_column_names*, *delimiter=None*)

Compacts data from multiple rows based on cell data.

Parameters *composite_key_column_names* : list

name of the user column to be used as keys for unflattening.

delimiter : unicode (default=None)

separator for the data in the result columns. Default is comma (,).

Returns : `_Unit`

Groups together cells in all columns (less the composite key) using “,” as string delimiter. The original rows are deleted. The grouping takes place based on a composite key passed as arguments.

class *Frame*

Large table of data.

Class with information about a large row and columnar data store in a frame, Has information needed to modify data and table structure.

Attributes

<code>column_names</code>	Column identifications in the current frame.
<code>name</code>	Set or get the name of the frame object.
<code>row_count</code>	Number of rows in the current frame.
<code>schema</code>	Current frame column names and types.
<code>status</code>	Current frame life cycle status.

Methods

<code>__init__(self[, source, name, _info])</code>	Create a Frame/frame.
<code>add_columns(self, func, schema[, columns_accessed])</code>	Add columns to current frame.
<code>append(self, data)</code>	Adds more data to the current frame.
<code>assign_sample(self, sample_percentages[, sample_labels, ...])</code>	Randomly group rows into user-defined classes.
<code>bin_column(self, column_name, cutoffs[, include_lowest, strict_binning, ...])</code>	Classify data into user-defined groups.
<code>bin_column_equal_depth(self, column_name[, num_bins, ...])</code>	Classify column into groups with the same frequency.
<code>bin_column_equal_width(self, column_name[, num_bins, ...])</code>	Classify column into same-width groups.
<code>categorical_summary(self, *column_inputs)</code>	[ALPHA] Compute a summary of the data in a column.
<code>classification_metrics(self, label_column, pred_column[, ...])</code>	Model statistics of accuracy, precision, and others.
<code>column_median(self, data_column[, weights_column])</code>	Calculate the (weighted) median of a column.
<code>column_mode(self, data_column[, weights_column, max_modes_returned])</code>	Evaluate the weights assigned to rows.
<code>column_summary_statistics(self, data_column[, ...])</code>	Calculate multiple statistics for a column.
<code>compute_misplaced_score(self, gravity)</code>	
<code>copy(self[, columns, where, name])</code>	Create new frame from current frame.
<code>correlation(self, data_column_names)</code>	Calculate correlation for two columns of current frame.
<code>correlation_matrix(self, data_column_names[, matrix_name])</code>	Calculate correlation matrix for two or more columns.
<code>count(self, where)</code>	Counts the number of rows which meet given criteria.

Table 25.3 – continued from previous page

<code>covariance(self, data_column_names)</code>	Calculate covariance for exactly two columns.
<code>covariance_matrix(self, data_column_names[, matrix_name])</code>	Calculate covariance matrix for two or more columns.
<code>cumulative_percent(self, sample_col)</code>	[BETA] Add column to frame with cumulative percent
<code>cumulative_sum(self, sample_col)</code>	[BETA] Add column to frame with cumulative percent
<code>dot_product(self, left_column_names, right_column_names, ...[, ...])</code>	[ALPHA] Calculate dot product for each row in current
<code>download(self[, n, offset, columns])</code>	Download a frame from the server into client workspace
<code>drop_columns(self, columns)</code>	Remove columns from the frame.
<code>drop_duplicates(self[, unique_columns])</code>	Modify the current frame, removing duplicate rows.
<code>drop_rows(self, predicate)</code>	Erase any row in the current frame which qualifies.
<code>ecdf(self, column[, result_frame_name])</code>	Builds new frame with columns for data and distribution
<code>entropy(self, data_column[, weights_column])</code>	Calculate the Shannon entropy of a column.
<code>export_to_csv(self, folder_name[, separator, count, offset])</code>	Write current frame to HDFS in csv format.
<code>export_to_hbase(self, table_name[, key_column_name, family_name])</code>	Write current frame to HBase table.
<code>export_to_hive(self, table_name)</code>	Write current frame to Hive table.
<code>export_to_jdbc(self, table_name[, connector_type, url, driver_name, ...])</code>	Write current frame to Jdbc table.
<code>export_to_json(self, folder_name[, count, offset])</code>	Write current frame to HDFS in JSON format.
<code>filter(self, predicate)</code>	Select all rows which satisfy a predicate.
<code>flatten_column(self, column[, delimiter])</code>	Spread data to multiple rows based on cell data.
<code>get_error_frame(self)</code>	Get a frame with error recordings.
<code>group_by(self, group_by_columns, *aggregation_arguments)</code>	[BETA] Create summarized frame.
<code>histogram(self, column_name[, num_bins, weight_column_name, bin_type])</code>	[BETA] Compute the histogram for a column in a frame
<code>inspect(self[, n, offset, columns, wrap, truncate, round, width, margin])</code>	Prints the frame data in readable format.
<code>join(self, right, left_on[, right_on, how, name])</code>	[BETA] Join operation on one or two frames, creating
<code>label_propagation(self, src_col_name, dest_col_name, ...[, ...])</code>	Label Propagation on Gaussian Random Fields.
<code>loadhbase(self, table_name, schema[, start_tag, end_tag])</code>	Append data from an hBase table into an existing (possi
<code>loadhive(self, query)</code>	Append data from a hive table into an existing (possi
<code>loadjdbc(self, table_name[, connector_type, url, driver_name, query])</code>	Append data from a Jdbc table into an existing (possi
<code>loopy_belief_propagation(self, src_col_name, ...[, ...])</code>	Message passing to infer state probabilities.
<code>quantiles(self, column_name, quantiles)</code>	New frame with Quantiles and their values.
<code>rename_columns(self, names)</code>	Rename columns
<code>sort(self, columns[, ascending])</code>	[BETA] Sort the data in a frame.
<code>sorted_k(self, k, column_names_and_ascending[, reduce_tree_depth])</code>	[ALPHA] Get a sorted subset of the data.
<code>take(self, n[, offset, columns])</code>	Get data subset.
<code>tally(self, sample_col, count_val)</code>	[BETA] Count number of times a value is seen.
<code>tally_percent(self, sample_col, count_val)</code>	[BETA] Compute a cumulative percent count.
<code>top_k(self, column_name, k[, weights_column])</code>	Most or least frequent column values.
<code>unflatten_column(self, composite_key_column_names[, delimiter])</code>	Compacts data from multiple rows based on cell data.

`__init__` (*self*, *source=None*, *name=None*)

Create a Frame/frame.

Parameters **source** : CsvFile | Frame (default=None)

A source of initial data.

name : str (default=None)

The name of the newly created frame. Default is None.

Notes

A frame with no name is subject to garbage collection.

If a string in the CSV file starts and ends with a double-quote (") character, the character is stripped off of the data before it is put into the field. Anything, including delimiters, between the double-quote characters is considered part of the str. If the first character after the delimiter is anything other than a double-quote character, the string will be composed of all the characters between the delimiters, including double-quotes. If the first field type is str, leading spaces on each row are considered part of the str. If the last field type is str, trailing spaces on each row are considered part of the str.

Examples

Create a new frame based upon the data described in the `CsvFile` object `my_csv_schema`. Name the frame "myframe". Create a Frame `my_frame` to access the data:

```
>>> my_frame = ta.Frame(my_csv_schema, "myframe")
```

A Frame object has been created and `my_frame` is its proxy. It brought in the data described by `my_csv_schema`. It is named `myframe`.

Create an empty frame; name it "yourframe":

```
>>> your_frame = ta.Frame(name='yourframe')
```

A frame has been created and Frame `your_frame` is its proxy. It has no data yet, but it does have the name `yourframe`.

25.4 *trustedanalytics* drop_frames

drop_frames (*items*)

Deletes the frame on the server.

Parameters **items** : [str | frame object | list [str | frame objects]]

Either the name of the frame object to delete or the frame object itself

25.5 *trustedanalytics* get_frame

get_frame (*identifier*)

Get handle to a frame object.

Parameters **identifier** : str | int

Name of the frame to get

Returns : Frame

frame object

25.6 *trustedanalytics* get_frame_names

get_frame_names ()

Retrieve names for all the frame objects on the server.

Returns : list

List of names

Global Methods

[drop_frames](#)

[get_frame](#)

[get_frame_names](#)

GRAPHS

Classes

26.1 *Graphs* Graph

26.1.1 *Graph* `__init__`

`__init__` (*self*, *name=None*)
<Missing Doc>

Parameters *name* : str (default=None)

Name for the new graph. Default is None.

26.1.2 *Graph* `fget`

`__private_ml`
Access to object's ml functionality (See `GraphMl`)

Parameters

Returns : `GraphMl`

`GraphMl` object

26.1.3 *Graph* `annotate_degrees`

`annotate_degrees` (*self*, *output_property_name*, *degree_option=None*, *input_edge_labels=None*)
Make new graph with degrees.

Parameters *output_property_name* : unicode

The name of the new property. The degree is stored in this property.

degree_option : unicode (default=None)

Indicator for the definition of degree to be used for the calculation. Permitted values:

- “out” (default value) : Degree is calculated as the out-degree.
- “in” : Degree is calculated as the in-degree.
- “undirected” : Degree is calculated as the undirected degree. (Assumes that the edges are all undirected.)

Any prefix of the strings “out”, “in”, “undirected” will select the corresponding option.

input_edge_labels : list (default=None)

If this list is provided, only edges whose labels are included in the given set will be considered in the degree calculation. In the default situation (when no list is provided), all edges will be used in the degree calculation, regardless of label.

Returns : dict

Dictionary containing the vertex type as the key and the corresponding vertex’s frame with a column storing the annotated degree for the vertex in a user specified property. Call `dictionary_name['label']` to get the handle to frame whose vertex type is label.

Creates a new graph which is the same as the input graph, with the addition that every vertex of the graph has its *degree* stored in a user-specified property.

Degree Calculation

A fundamental quantity in graph analyses is the degree of a vertex: The degree of a vertex is the number of edges adjacent to it.

For a directed edge relation, a vertex has both an out-degree (the number of edges leaving the vertex) and an in-degree (the number of edges entering the vertex).

The toolkit provides this routine for calculating the degrees of vertices. This calculation could be performed with a Gremlin query on smaller datasets because Gremlin queries cannot be executed on a distributed scale. The Trusted Analytics routine `annotate_degrees` can be executed at distributed scale.

In the presence of edge weights, vertices can have weighted degrees: The weighted degree of a vertex is the sum of weights of edges adjacent to it. Analogously, the weighted in-degree of a vertex is the sum of the weights of the edges entering it, and the weighted out-degree is the sum of the weights of the edges leaving the vertex.

The toolkit provides `annotate_weighted_degrees` for the distributed calculation of weighted vertex degrees.

26.1.4 Graph `annotate_weighted_degrees`

```
annotate_weighted_degrees (self,      output_property_name,      degree_option=None,      in-put_edge_labels=None,      edge_weight_property=None,      edge_weight_default=None)
```

Calculates the weighted degree of each vertex with respect to an (optional) set of labels.

Parameters **output_property_name** : unicode

property name of where to store output

degree_option : unicode (default=None)

choose from ‘out’, ‘in’, ‘undirected’

input_edge_labels : list (default=None)

labels of edge types that should be included

edge_weight_property : unicode (default=None)

property name of edge weight, if not provided all edges are weighted equally

edge_weight_default : float64 (default=None)

default edge weight

Returns : dict

Pulls graph from underlying store, calculates weighted degrees and writes them into the property specified, and then writes the output graph to the underlying store.

Degree Calculation

A fundamental quantity in graph analyses is the degree of a vertex: The degree of a vertex is the number of edges adjacent to it.

For a directed edge relation, a vertex has both an out-degree (the number of edges leaving the vertex) and an in-degree (the number of edges entering the vertex).

The toolkit provides a routine *annotate_degrees* for calculating the degrees of vertices. This calculation could be performed with a Gremlin query on smaller datasets because Gremlin queries cannot be executed on a distributed scale. The Trusted Analytics routine *annotate_degrees* can be executed at distributed scale.

In the presence of edge weights, vertices can have weighted degrees: The weighted degree of a vertex is the sum of weights of edges adjacent to it. Analogously, the weighted in-degree of a vertex is the sum of the weights of the edges entering it, and the weighted out-degree is the sum of the weights of the edges leaving the vertex.

The toolkit provides this routine for the distributed calculation of weighted vertex degrees.

26.1.5 Graph clustering_coefficient

clustering_coefficient (*self*, *output_property_name=None*, *input_edge_labels=None*)

Coefficient of graph with respect to labels.

Parameters **output_property_name** : unicode (default=None)

The name of the new property to which each vertex's local clustering coefficient will be written. If this option is not specified, no output frame will be produced and only the global clustering coefficient will be returned.

input_edge_labels : list (default=None)

If this list is provided, only edges whose labels are included in the given set will be considered in the clustering coefficient calculation. In the default situation (when no list is provided), all edges will be used in the calculation, regardless of label. It is required that all edges that enter into the clustering coefficient analysis be undirected.

Returns : dict

Dictionary of the global clustering coefficient of the graph or, if local clustering coefficients are requested, a reference to the frame with local clustering coefficients stored at properties at each vertex.

Calculates the clustering coefficient of the graph with respect to an (optional) set of labels.

Pulls graph from underlying store, calculates degrees and writes them into the property specified, and then writes the output graph to the underlying store.

Warning: THIS FUNCTION IS FOR UNDIRECTED GRAPHS. If it is called on a directed graph, its output is NOT guaranteed to calculate the local directed clustering coefficients.

Clustering Coefficients

The clustering coefficient of a graph provides a measure of how tightly clustered an undirected graph is. Informally, if the edge relation denotes “friendship”, the clustering coefficient of the graph is the probability that two people are friends given that they share a common friend.

More formally:

$$cc(G) = \frac{\|\{(u, v, w) \in V^3 : \{u, v\}, \{u, w\}, \{v, w\} \in E\}\|}{\|\{(u, v, w) \in V^3 : \{u, v\}, \{u, w\} \in E\}\|}$$

Analogously, the clustering coefficient of a vertex provides a measure of how tightly clustered that vertex’s neighborhood is. Informally, if the edge relation denotes “friendship”, the clustering coefficient at a vertex v is the probability that two acquaintances of v are themselves friends.

More formally:

$$cc(v) = \frac{\|\{(u, v, w) \in V^3 : \{u, v\}, \{u, w\}, \{v, w\} \in E\}\|}{\|\{(u, v, w) \in V^3 : \{v, u\}, \{v, w\} \in E\}\|}$$

The toolkit provides the function `clustering_coefficient` which computes both local and global clustering coefficients for a given undirected graph.

For more details on the mathematics and applications of clustering coefficients, see http://en.wikipedia.org/wiki/Clustering_coefficient.

26.1.6 Graph copy

copy (*self*, *name=None*)

Make a copy of the current graph.

Parameters **name** : unicode (default=None)

The name for the copy of the graph. Default is None.

Returns : dict

A copy of the original graph.

26.1.7 *Graph* define_edge_type

define_edge_type (*self*, *label*, *src_vertex_label*, *dest_vertex_label*, *directed=False*)

Define an edge type.

Parameters **label** : unicode

Label of the edge type.

src_vertex_label : unicode

The src “type” of vertices this edge connects.

dest_vertex_label : unicode

The destination “type” of vertices this edge connects.

directed : bool (default=False)

True if edges are directed, false if they are undirected.

Returns : _Unit

26.1.8 *Graph* define_vertex_type

define_vertex_type (*self*, *label*)

Define a vertex type by label.

Parameters **label** : unicode

Label of the vertex type.

Returns : _Unit

26.1.9 *Graph* edge_count

edge_count

Get the total number of edges in the graph.

Parameters

Returns : int

Total number of edges in the graph

int32 The number of edges in the graph.

Examples

```
>>> my_graph.edge_count
```

The result given is:

1194

26.1.10 *Graph* edges

edges

Edge frame collection

Parameters

Examples

Inspect edges with the supplied label:

```
>>> my_graph.edges['label'].inspect()
```

26.1.11 *Graph* export_to_titan

export_to_titan (*self*, *new_graph_name=None*)

Convert current graph to TitanGraph.

Parameters **new_graph_name** : unicode (default=None)

The name of the new graph. Default is None.

Returns : dict

A new TitanGraph.

Convert this Graph into a TitanGraph object. This will be a new graph backed by Titan with all of the data found in this graph.

26.1.12 *Graph* graphx_connected_components

graphx_connected_components (*self*, *output_property*)

Implements the connected components computation on a graph by invoking graphx api.

Parameters **output_property** : unicode

The name of the column containing the connected component value.

Returns : dict

Dictionary containing the vertex type as the key and the corresponding vertex's frame with a connected component column. Call `dictionary_name['label']` to get the handle to frame whose vertex type is label.

Pulls graph from underlying store, sends it off to the ConnectedComponentGraphXDefault, and then writes the output graph back to the underlying store.

Connected Components (CC)

Connected components are disjoint subgraphs in which all vertices are connected to all other vertices in the same component via paths, but not connected via paths to vertices in any other component. The connected components algorithm uses message passing along a specified edge type to find all of the connected components of a graph and label each edge with the identity of the component to which it belongs. The algorithm is specific to an edge type, hence in graphs with several different types of edges, there may be multiple, overlapping sets of connected components.

The algorithm works by assigning each vertex a unique numerical index and passing messages between neighbors. Vertices pass their indices back and forth with their neighbors and update their own index as the minimum of their current index and all other indices received. This algorithm continues until there is no change in any of the vertex indices. At the end of the algorithm, the unique levels of the indices denote the distinct connected components. The complexity of the algorithm is proportional to the diameter of the graph.

26.1.13 Graph `graphx_pagerank`

graphx_pagerank (*self*, *output_property*, *input_edge_labels=None*, *max_iterations=None*, *reset_probability=None*, *convergence_tolerance=None*)

Determine which vertices are the most important.

Parameters *output_property* : unicode

Name of the property to which pagerank value will be stored on vertex and edge.

input_edge_labels : list (default=None)

List of edge labels to consider for pagerank computation. Default is all edges are considered.

max_iterations : int32 (default=None)

The maximum number of iterations that will be invoked. The valid range is all positive int. Invalid value will terminate with vertex page rank set to `reset_probability`. Default is 20.

reset_probability : float64 (default=None)

The probability that the random walk of a page is reset. Default is 0.15.

convergence_tolerance : float64 (default=None)

The amount of change in cost function that will be tolerated at convergence. If this parameter is specified, `max_iterations` is not considered as a stopping condition. If the change is less than this threshold, the algorithm exits earlier. The valid value range is all float and zero. Default is 0.001.

Returns : dict

dict((vertex_dictionary, (label, Frame)), (edge_dictionary, (label, Frame))).

Dictionary containing dictionaries of labeled vertices and labeled edges.

For the *vertex_dictionary* the vertex type is the key and the corresponding vertex's frame with a new column storing the page rank value for the vertex. Call `vertex_dictionary['label']` to get the handle to frame whose vertex type is label.

For the *edge_dictionary* the edge type is the key and the corresponding edge's frame with a new column storing the page rank value for the edge. Call `edge_dictionary['label']` to get the handle to frame whose edge type is label.

Pulls graph from underlying store, sends it off to the PageRankRunner, and then writes the output graph back to the underlying store.

This method (currently) only supports Titan for graph storage.

**** Experimental Feature ****

Basics and Background

PageRank is a method for determining which vertices in a directed graph are the most central or important. *PageRank* gives each vertex a score which can be interpreted as the probability that a person randomly walking along the edges of the graph will visit that vertex.

The calculation of *PageRank* is based on the supposition that if a vertex has many vertices pointing to it, then it is “important”, and that a vertex grows in importance as more important vertices point to it. The calculation is based only on the network structure of the graph and makes no use of any side data, properties, user-provided scores or similar non-topological information.

PageRank was most famously used as the core of the Google search engine for many years, but as a general measure of *centrality* in a graph, it has other uses to other problems, such as *recommendation systems* and analyzing predator-prey food webs to predict extinctions.

Background references

- Basic description and principles: [Wikipedia: PageRank¹](#)
- Applications to food web analysis: [Stanford: Applications of PageRank²](#)
- Applications to recommendation systems: [PLOS: Computational Biology³](#)

Mathematical Details of PageRank Implementation

The Trusted Analytics implementation of *PageRank* satisfies the following equation at each vertex v of the graph:

$$PR(v) = \frac{\rho}{n} + \rho \left(\sum_{u \in InSet(v)} \frac{PR(u)}{L(u)} \right)$$

Where:

- v — a vertex
- $L(v)$ — outbound degree of the vertex v
- $PR(v)$ — *PageRank* score of the vertex v
- $InSet(v)$ — set of vertices pointing to the vertex v
- n — total number of vertices in the graph
- ρ — user specified damping factor (also known as reset probability)

Termination is guaranteed by two mechanisms.

- The user can specify a convergence threshold so that the algorithm will terminate when, at every vertex, the difference between successive approximations to the *PageRank* score falls below the convergence threshold.
- The user can specify a maximum number of iterations after which the algorithm will terminate.

¹<http://en.wikipedia.org/wiki/PageRank>

²<http://web.stanford.edu/class/msande233/handouts/lecture8.pdf>

³<http://www.ploscompbiol.org/article/fetchObject.action?uri=info%3Adoi%2F10.1371%2Fjournal.pcbi.1000494&representation=PDF>

26.1.14 *Graph* graphx_triangle_count

graphx_triangle_count (*self*, *output_property*, *input_edge_labels=None*)

Number of triangles among vertices of current graph.

Parameters **output_property** : unicode

The name of output property to be added to vertex/edge upon completion.

input_edge_labels : list (default=None)

The name of edge labels to be considered for triangle count. Default is all edges are considered.

Returns : dict

dict(label, Frame).

Dictionary containing the vertex type as the key and the corresponding vertex's frame with a triangle_count column. Call dictionary_name['label'] to get the handle to frame whose vertex type is label.

**** Experimental Feature ****

Counts the number of triangles among vertices in an undirected graph. If an edge is marked bidirectional, the implementation opts for canonical orientation of edges hence counting it only once (similar to an undirected graph).

26.1.15 *GraphML* belief_propagation

ml.belief_propagation (*self*, *prior_property*, *posterior_property*, *edge_weight_property=None*, *convergence_threshold=None*, *max_iterations=None*)

Classification on sparse data using Belief Propagation.

Parameters **prior_property** : unicode

Name of the vertex property which contains the prior belief for the vertex.

posterior_property : unicode

Name of the vertex property which will contain the posterior belief for each vertex.

edge_weight_property : unicode (default=None)

Name of the edge property that contains the edge weight for each edge.

convergence_threshold : float64 (default=None)

Belief propagation will terminate when the average change in posterior beliefs between supersteps is less than or equal to this threshold.

max_iterations : int32 (default=None)

The maximum number of supersteps that the algorithm will execute. The valid range is all positive int.

Returns : dict

Progress report for belief propagation in the format of a multiple-line string.

Belief propagation by the sum-product algorithm. This algorithm analyzes a graphical model with prior beliefs using sum product message passing. The priors are read from a property in the graph, the posteriors are written to another property in the graph. This is the GraphX-based implementation of belief propagation.

See [Loopy Belief Propagation](#) for a more in-depth discussion of BP and LBP.

26.1.16 *GraphML* `kclique_percolation`

`ml.kclique_percolation(self, clique_size, community_property_label)`
 [ALPHA] Find groups of vertices with similar attributes.

Parameters `clique_size` : int32

The sizes of the cliques used to form communities. Larger values of clique size result in fewer, smaller communities that are more connected. Must be at least 2.

community_property_label : unicode

Name of the community property of vertex that will be updated/created in the graph. This property will contain for each vertex the set of communities that contain that vertex.

Returns : dict

Dictionary of vertex label and frame, Execution time.

Community Detection Using the K-Clique Percolation Algorithm

Overview

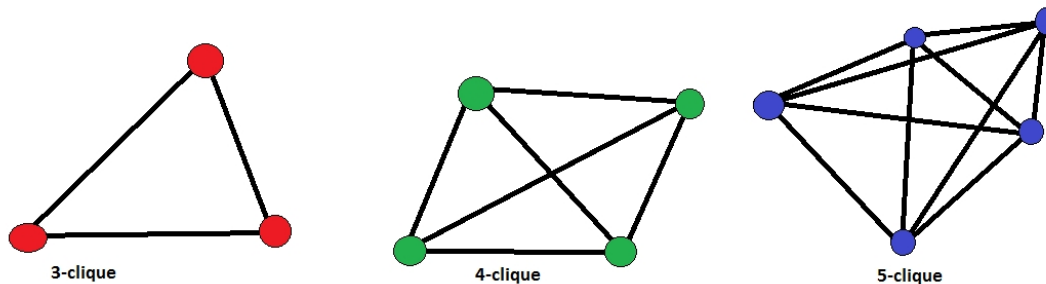
Modeling data as a graph captures relations, for example, friendship ties between social network users or chemical interactions between proteins. Analyzing the structure of the graph reveals collections (often termed ‘communities’) of vertices that are more likely to interact amongst each other. Examples could include a community of friends in a social network or a collection of highly interacting proteins in a cellular process.

Trusted Analytics provides community detection using the k-Clique percolation method first proposed by Palla et. al. [R1] that has been widely used in many contexts.

K-Clique Percolation

K-clique percolation is a method for detecting community structure in graphs. Here we provide mathematical background on how communities are defined in the context of the k-clique percolation algorithm.

A clique is a group of vertices in which every vertex is connected (via undirected edge) with every other vertex in the clique. This graphically looks like a triangle or a structure composed of triangles:

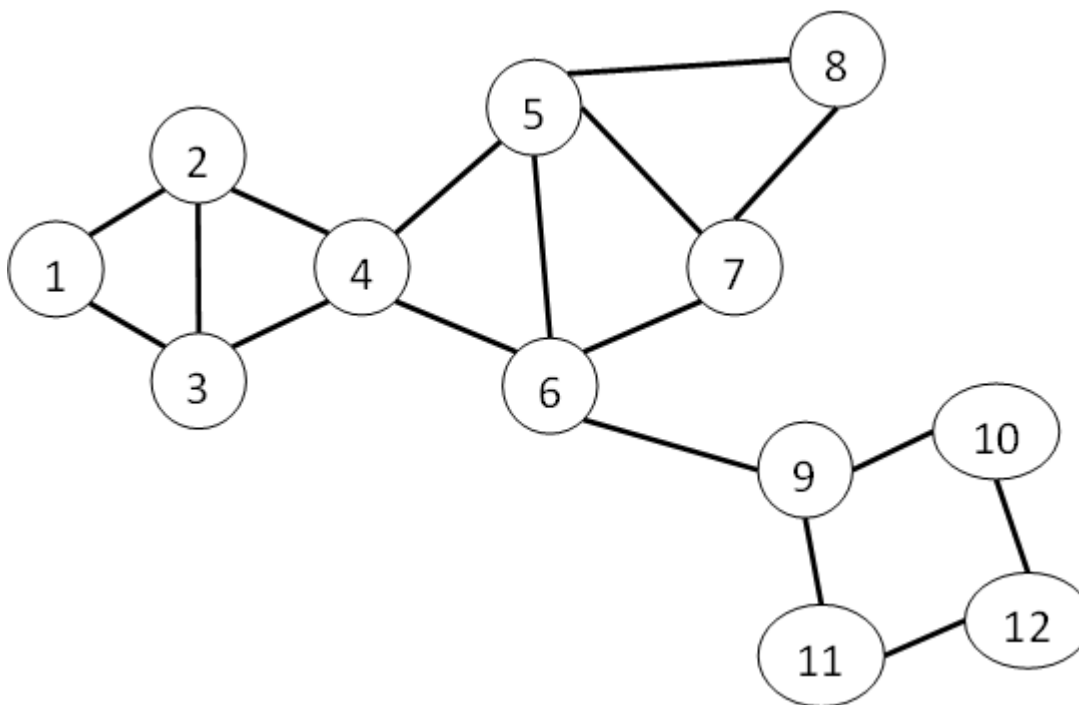


A clique is certainly a community in the sense that its vertices are all connected, but, it is too restrictive for most purposes, since it is natural some members of a community may not interact.

Mathematically, a k -clique has k vertices, each with $k - 1$ common edges, each of which connects to another vertex in the k -clique. The k -clique percolation method forms communities by taking unions of k -cliques that have $k - 1$ vertices in common.

K-Clique Example

In the graph below, the cliques are the sections defined by their triangular appearance and the 3-clique communities are $\{1, 2, 3, 4\}$ and $\{4, 5, 6, 7, 8\}$. The vertices 9, 10, 11, 12 are not in 3-cliques, therefore they do not belong to any community. Vertex 4 belongs to two distinct (but overlapping) communities.



Distributed Implementation of K-Clique Community Detection

The implementation of k -clique community detection in Trusted Analytics is a fully distributed implementation that follows the map-reduce algorithm proposed in Varamesh et. al. [R2] .

It has the following steps:

1. All k -cliques are *enumerated*.
2. k -cliques are used to build a “clique graph” by declaring each k -clique to be a vertex in a new graph and placing edges between k -cliques that share $k-1$ vertices in the base graph.
3. A *connected component* analysis is performed on the clique graph. Connected components of the clique graph correspond to k -clique communities in the base graph.
4. The connected components information for the clique graph is projected back down to the base graph, providing each vertex with the set of k -clique communities to which it belongs.

Notes

Spawns a number of Spark jobs that cannot be calculated before execution (it is bounded by the diameter of the clique graph derived from the input graph). For this reason, the initial loading, clique enumeration and clique-graph construction steps are tracked with a single progress bar (this is most of the time), and then successive iterations of analysis of the clique graph are tracked with many short-lived progress bars, and then finally the result is written out.

26.1.17 *Graph name*

name

Set or get the name of the graph object.

Parameters

Change or retrieve graph object identification. Identification names must start with a letter and are limited to alphanumeric characters and the `_` character.

Examples

```
>>> my_graph.name
"csv_data"

>>> my_graph.name = "cleaned_data"
>>> my_graph.name
"cleaned_data"
```

26.1.18 *Graph status*

status

Current graph life cycle status.

Parameters

Returns : str

Status of the graph

One of three statuses: Active, Deleted, Deleted_Final Active: available for use Deleted: has been scheduled for deletion Deleted_Final: backend files have been removed from disk.

26.1.19 *Graph vertex_count*

vertex_count

Get the total number of vertices in the graph.

Parameters

Returns int32

The number of vertices in the graph.

Examples


```
>>> my_graph.vertex_count
```

The result given is:

```
1194
```

26.1.20 *Graph* vertices

vertices

Vertex frame collection

Parameters

Examples

Inspect vertices with the supplied label:

```
>>> my_graph.vertices['label'].inspect()
```

class Graph

Creates a seamless property graph.

A seamless graph is a collection of vertex and edge lists stored as frames. This allows frame-like operations against graph data. Many frame methods are available to work with vertices and edges. Vertex and edge properties are stored as columns.

A seamless graph is better suited for bulk *OLAP*-type operations whereas a Titan graph is better suited to *OLTP*.

Attributes

<code>edge_count</code>	Get the total number of edges in the graph.
<code>edges</code>	Edge frame collection
<code>name</code>	Set or get the name of the graph object.
<code>status</code>	Current graph life cycle status.
<code>vertex_count</code>	Get the total number of vertices in the graph.
<code>vertices</code>	Vertex frame collection

Methods

<code>__init__(self[, name, _info])</code>	<Missing Doc>
<code>__init__(self, entity)</code>	<Missing Doc>
<code>annotate_degrees(self, output_property_name[, degree_option, ...])</code>	Make new graph with degrees.
<code>annotate_weighted_degrees(self, output_property_name[, ...])</code>	Calculates the weighted degree of each vertex with respect to an (optional) set of labels.
<code>clustering_coefficient(self, output_property_name, ...)</code>	Coefficient of graph with respect to labels.
<code>copy(self[, name])</code>	Make a copy of the current graph.
<code>define_edge_type(self, label, src_vertex_label, dest_vertex_label)</code>	Define an edge type.
<code>define_vertex_type(self, label)</code>	Define a vertex type by label.
<code>export_to_titan(self[, new_graph_name])</code>	Convert current graph to TitanGraph.
<code>graphx_connected_components(self, output_property)</code>	Implements the connected components computation on a graph by invoking graphx api.
<code>graphx_pagerank(self, output_property[, input_edge_labels, ...])</code>	Determine which vertices are the most important.
<code>graphx_triangle_count(self, output_property[, input_edge_labels])</code>	Number of triangles among vertices of current graph.
<code>ml.belief_propagation(self, prior_property, posterior_property)</code>	Classification on sparse data using Belief Propagation.
<code>ml.kclique_percolation(self, clique_size, ...)</code>	[ALPHA] Find groups of vertices with similar attributes.

`__init__(self, name=None)`

<Missing Doc>

Parameters `name` : str (default=None)

Name for the new graph. Default is None.

26.2 Graphs TitanGraph

26.2.1 TitanGraph __init__

`__init__(self, name=None)`

Initialize the graph.

Parameters `name` : (default=None)

26.2.2 TitanGraph fget

`__private_ml`

Access to object's ml functionality (See TitanGraphMl)

Parameters

Returns : TitanGraphMI
TitanGraphMI object

26.2.3 *TitanGraph* fget

__private_query
Access to object's query functionality (See `TitanGraphQuery`)

Parameters

Returns : TitanGraphQuery
TitanGraphQuery object

26.2.4 *TitanGraph* annotate_degrees

annotate_degrees (*self*, *output_property_name*, *degree_option=None*, *input_edge_labels=None*)
Make new graph with degrees.

Parameters **output_property_name** : unicode

The name of the new property. The degree is stored in this property.

degree_option : unicode (default=None)

Indicator for the definition of degree to be used for the calculation. Permitted values:

- “out” (default value) : Degree is calculated as the out-degree.
- “in” : Degree is calculated as the in-degree.
- “undirected” : Degree is calculated as the undirected degree. (Assumes that the edges are all undirected.)

Any prefix of the strings “out”, “in”, “undirected” will select the corresponding option.

input_edge_labels : list (default=None)

If this list is provided, only edges whose labels are included in the given set will be considered in the degree calculation. In the default situation (when no list is provided), all edges will be used in the degree calculation, regardless of label.

Returns : dict

Dictionary containing the vertex type as the key and the corresponding vertex's frame with a column storing the annotated degree for the vertex in a user specified property. Call `dictionary_name['label']` to get the handle to frame whose vertex type is label.

Creates a new graph which is the same as the input graph, with the addition that every vertex of the graph has its *degree* stored in a user-specified property.

Degree Calculation

A fundamental quantity in graph analyses is the degree of a vertex: The degree of a vertex is the number of edges adjacent to it.

For a directed edge relation, a vertex has both an out-degree (the number of edges leaving the vertex) and an in-degree (the number of edges entering the vertex).

The toolkit provides this routine for calculating the degrees of vertices. This calculation could be performed with a Gremlin query on smaller datasets because Gremlin queries cannot be executed on a distributed scale. The Trusted Analytics routine `annotate_degrees` can be executed at distributed scale.

In the presence of edge weights, vertices can have weighted degrees: The weighted degree of a vertex is the sum of weights of edges adjacent to it. Analogously, the weighted in-degree of a vertex is the sum of the weights of the edges entering it, and the weighted out-degree is the sum of the weights of the edges leaving the vertex.

The toolkit provides `annotate_weighted_degrees` for the distributed calculation of weighted vertex degrees.

26.2.5 *TitanGraph* `annotate_weighted_degrees`

```
annotate_weighted_degrees (self,      output_property_name,      degree_option=None,      in-put_edge_labels=None,      edge_weight_property=None,      edge_weight_default=None)
```

Calculates the weighted degree of each vertex with respect to an (optional) set of labels.

Parameters `output_property_name` : unicode

property name of where to store output

`degree_option` : unicode (default=None)

choose from 'out', 'in', 'undirected'

`input_edge_labels` : list (default=None)

labels of edge types that should be included

`edge_weight_property` : unicode (default=None)

property name of edge weight, if not provided all edges are weighted equally

`edge_weight_default` : float64 (default=None)

default edge weight

Returns : dict

Pulls graph from underlying store, calculates weighted degrees and writes them into the property specified, and then writes the output graph to the underlying store.

Degree Calculation

A fundamental quantity in graph analyses is the degree of a vertex: The degree of a vertex is the number of edges adjacent to it.

For a directed edge relation, a vertex has both an out-degree (the number of edges leaving the vertex) and an in-degree (the number of edges entering the vertex).

The toolkit provides a routine `annotate_degrees` for calculating the degrees of vertices. This calculation could be performed with a Gremlin query on smaller datasets because Gremlin queries cannot be executed on a distributed scale. The Trusted Analytics routine `annotate_degrees` can be executed at distributed scale.

In the presence of edge weights, vertices can have weighted degrees: The weighted degree of a vertex is the sum of weights of edges adjacent to it. Analogously, the weighted in-degree of a vertex is the sum of the weights of the edges entering it, and the weighted out-degree is the sum of the weights of the edges leaving the vertex.

The toolkit provides this routine for the distributed calculation of weighted vertex degrees.

26.2.6 TitanGraph clustering_coefficient

clustering_coefficient (*self*, *output_property_name=None*, *input_edge_labels=None*)

Coefficient of graph with respect to labels.

Parameters **output_property_name** : unicode (default=None)

The name of the new property to which each vertex's local clustering coefficient will be written. If this option is not specified, no output frame will be produced and only the global clustering coefficient will be returned.

input_edge_labels : list (default=None)

If this list is provided, only edges whose labels are included in the given set will be considered in the clustering coefficient calculation. In the default situation (when no list is provided), all edges will be used in the calculation, regardless of label. It is required that all edges that enter into the clustering coefficient analysis be undirected.

Returns : dict

Dictionary of the global clustering coefficient of the graph or, if local clustering coefficients are requested, a reference to the frame with local clustering coefficients stored at properties at each vertex.

Calculates the clustering coefficient of the graph with respect to an (optional) set of labels.

Pulls graph from underlying store, calculates degrees and writes them into the property specified, and then writes the output graph to the underlying store.

Warning: THIS FUNCTION IS FOR UNDIRECTED GRAPHS. If it is called on a directed graph, its output is NOT guaranteed to calculate the local directed clustering coefficients.

Clustering Coefficients

The clustering coefficient of a graph provides a measure of how tightly clustered an undirected graph is. Informally, if the edge relation denotes “friendship”, the clustering coefficient of the graph is the probability that two people are friends given that they share a common friend.

More formally:

$$cc(G) = \frac{\|\{(u, v, w) \in V^3 : \{u, v\}, \{u, w\}, \{v, w\} \in E\}\|}{\|\{(u, v, w) \in V^3 : \{u, v\}, \{u, w\} \in E\}\|}$$

Analogously, the clustering coefficient of a vertex provides a measure of how tightly clustered that vertex's neighborhood is. Informally, if the edge relation denotes “friendship”, the clustering coefficient at a vertex v is the probability that two acquaintances of v are themselves friends.

More formally:

$$cc(v) = \frac{\|\{(u, v, w) \in V^3 : \{u, v\}, \{u, w\}, \{v, w\} \in E\}\|}{\|\{(u, v, w) \in V^3 : \{v, u\}, \{v, w\} \in E\}\|}$$

The toolkit provides the function `clustering_coefficient` which computes both local and global clustering coefficients for a given undirected graph.

For more details on the mathematics and applications of clustering coefficients, see http://en.wikipedia.org/wiki/Clustering_coefficient.

26.2.7 *TitanGraph* copy

copy (*self*, *name=None*)

Make a copy of the current graph.

Parameters **name** : unicode (default=None)

The name for the copy of the graph. Default is None.

Returns : dict

A copy of the original graph.

26.2.8 *TitanGraph* export_to_graph

export_to_graph (*self*)

Export from ta.TitanGraph to ta.Graph.

Parameters

Returns : dict

26.2.9 *TitanGraph* graph_clustering

graph_clustering (*self*, *edge_distance*)

Performs graph clustering over an initial titan graph.

Parameters **edge_distance** : unicode

Column name for the edge distance.

Returns : _Unit

Performs graph clustering over an initial titan graph using a distributed edge collapse algorithm.

26.2.10 *TitanGraph* graphx_connected_components

graphx_connected_components (*self*, *output_property*)

Implements the connected components computation on a graph by invoking graphx api.

Parameters **output_property** : unicode

The name of the column containing the connected component value.

Returns : dict

Dictionary containing the vertex type as the key and the corresponding vertex's frame with a connected component column. Call `dictionary_name['label']` to get the handle to frame whose vertex type is label.

Pulls graph from underlying store, sends it off to the `ConnectedComponentGraphXDefault`, and then writes the output graph back to the underlying store.

Connected Components (CC)

Connected components are disjoint subgraphs in which all vertices are connected to all other vertices in the same component via paths, but not connected via paths to vertices in any other component. The connected components algorithm uses message passing along a specified edge type to find all of the connected components of a graph and label each edge with the identity of the component to which it belongs. The algorithm is specific to an edge type, hence in graphs with several different types of edges, there may be multiple, overlapping sets of connected components.

The algorithm works by assigning each vertex a unique numerical index and passing messages between neighbors. Vertices pass their indices back and forth with their neighbors and update their own index as the minimum of their current index and all other indices received. This algorithm continues until there is no change in any of the vertex indices. At the end of the algorithm, the unique levels of the indices denote the distinct connected components. The complexity of the algorithm is proportional to the diameter of the graph.

26.2.11 *TitanGraph* `graphx_pagerank`

graphx_pagerank (*self*, *output_property*, *input_edge_labels=None*, *max_iterations=None*, *reset_probability=None*, *convergence_tolerance=None*)

Determine which vertices are the most important.

Parameters **output_property** : unicode

Name of the property to which pagerank value will be stored on vertex and edge.

input_edge_labels : list (default=None)

List of edge labels to consider for pagerank computation. Default is all edges are considered.

max_iterations : int32 (default=None)

The maximum number of iterations that will be invoked. The valid range is all positive int. Invalid value will terminate with vertex page rank set to `reset_probability`. Default is 20.

reset_probability : float64 (default=None)

The probability that the random walk of a page is reset. Default is 0.15.

convergence_tolerance : float64 (default=None)

The amount of change in cost function that will be tolerated at convergence. If this parameter is specified, `max_iterations` is not considered as a stopping condition. If the change is less than this threshold, the algorithm exits earlier. The valid value range is all float and zero. Default is 0.001.

Returns : dict

dict((vertex_dictionary, (label, Frame)), (edge_dictionary, (label, Frame))).

Dictionary containing dictionaries of labeled vertices and labeled edges.

For the *vertex_dictionary* the vertex type is the key and the corresponding vertex's frame with a new column storing the page rank value for the vertex. Call `vertex_dictionary['label']` to get the handle to frame whose vertex type is label.

For the *edge_dictionary* the edge type is the key and the corresponding edge's frame with a new column storing the page rank value for the edge. Call `edge_dictionary['label']` to get the handle to frame whose edge type is label.

Pulls graph from underlying store, sends it off to the PageRankRunner, and then writes the output graph back to the underlying store.

This method (currently) only supports Titan for graph storage.

**** Experimental Feature ****

Basics and Background

PageRank is a method for determining which vertices in a directed graph are the most central or important. *PageRank* gives each vertex a score which can be interpreted as the probability that a person randomly walking along the edges of the graph will visit that vertex.

The calculation of *PageRank* is based on the supposition that if a vertex has many vertices pointing to it, then it is “important”, and that a vertex grows in importance as more important vertices point to it. The calculation is based only on the network structure of the graph and makes no use of any side data, properties, user-provided scores or similar non-topological information.

PageRank was most famously used as the core of the Google search engine for many years, but as a general measure of *centrality* in a graph, it has other uses to other problems, such as *recommendation systems* and analyzing predator-prey food webs to predict extinctions.

Background references

- Basic description and principles: [Wikipedia: PageRank](#)⁴
- Applications to food web analysis: [Stanford: Applications of PageRank](#)⁵
- Applications to recommendation systems: [PLOS: Computational Biology](#)⁶

Mathematical Details of PageRank Implementation

The Trusted Analytics implementation of *PageRank* satisfies the following equation at each vertex v of the graph:

$$PR(v) = \frac{\rho}{n} + \rho \left(\sum_{u \in InSet(v)} \frac{PR(u)}{L(u)} \right)$$

Where:

v — a vertex

$L(v)$ — outbound degree of the vertex v

$PR(v)$ — *PageRank* score of the vertex v

$InSet(v)$ — set of vertices pointing to the vertex v

n — total number of vertices in the graph

⁴<http://en.wikipedia.org/wiki/PageRank>

⁵<http://web.stanford.edu/class/msande233/handouts/lecture8.pdf>

⁶<http://www.ploscompbiol.org/article/fetchObject.action?uri=info%3Adoi%2F10.1371%2Fjournal.pcbi.1000494&representation=PDF>

ρ — user specified damping factor (also known as reset probability)

Termination is guaranteed by two mechanisms.

- The user can specify a convergence threshold so that the algorithm will terminate when, at every vertex, the difference between successive approximations to the *PageRank* score falls below the convergence threshold.
- The user can specify a maximum number of iterations after which the algorithm will terminate.

26.2.12 *TitanGraph* graphx_triangle_count

graphx_triangle_count (*self*, *output_property*, *input_edge_labels=None*)

Number of triangles among vertices of current graph.

Parameters **output_property** : unicode

The name of output property to be added to vertex/edge upon completion.

input_edge_labels : list (default=None)

The name of edge labels to be considered for triangle count. Default is all edges are considered.

Returns : dict

dict(label, Frame).

Dictionary containing the vertex type as the key and the corresponding vertex's frame with a triangle_count column. Call dictionary_name['label'] to get the handle to frame whose vertex type is label.

**** Experimental Feature ****

Counts the number of triangles among vertices in an undirected graph. If an edge is marked bidirectional, the implementation opts for canonical orientation of edges hence counting it only once (similar to an undirected graph).

26.2.13 *TitanGraphMI* belief_propagation

ml.belief_propagation (*self*, *prior_property*, *posterior_property*, *edge_weight_property=None*, *convergence_threshold=None*, *max_iterations=None*)

Classification on sparse data using Belief Propagation.

Parameters **prior_property** : unicode

Name of the vertex property which contains the prior belief for the vertex.

posterior_property : unicode

Name of the vertex property which will contain the posterior belief for each vertex.

edge_weight_property : unicode (default=None)

Name of the edge property that contains the edge weight for each edge.

convergence_threshold : float64 (default=None)

Belief propagation will terminate when the average change in posterior beliefs between supersteps is less than or equal to this threshold.

max_iterations : int32 (default=None)

The maximum number of supersteps that the algorithm will execute. The valid range is all positive int.

Returns : dict

Progress report for belief propagation in the format of a multiple-line string.

Belief propagation by the sum-product algorithm. This algorithm analyzes a graphical model with prior beliefs using sum product message passing. The priors are read from a property in the graph, the posteriors are written to another property in the graph. This is the GraphX-based implementation of belief propagation.

See *Loopy Belief Propagation* for a more in-depth discussion of BP and LBP.

26.2.14 *TitanGraph* name

name

Set or get the name of the graph object.

Parameters

Change or retrieve graph object identification. Identification names must start with a letter and are limited to alphanumeric characters and the `_` character.

Examples

```
>>> my_graph.name
"csv_data"

>>> my_graph.name = "cleaned_data"
>>> my_graph.name
"cleaned_data"
```

26.2.15 *TitanGraphQuery* gremlin

`query.gremlin(self, gremlin)`

Executes a Gremlin query.

Parameters **gremlin** : unicode

The Gremlin script to execute.

Examples of Gremlin queries:

`g.V[0..9]` - Returns the first 10 vertices in graph
`g.V.userId` - Returns the userId property from vertices
`g.V('name','hercules').out('father').out('father').name` - Returns the name of Hercules' grandfather

Returns : dict

List of query results serialized to JSON and runtime of Gremlin query in seconds. The list of results is in GraphSON format(for vertices or edges) or JSON (for other results like counts). GraphSON is a JSON-based format for property graphs which uses reserved keys that begin with underscores to encode vertex and edge metadata.

Examples of valid GraphSON:

```
{ \"name\": \"lop\", \"lang\": \"java\", \"_id\": \"3\", \"_type\": \"vertex\" }
{ \"weight\": 1, \"_id\": \"8\", \"_type\": \"edge\", \"_outV\": \"1\", \"_inV\": \"4\" }
```

See

<https://github.com/tinkerpop/blueprints/wiki/GraphSON-Reader-and-Writer-Library>

Executes a Gremlin query on an existing graph.

Notes

The query does not support pagination so the results of query should be limited using the Gremlin range filter [i..j], for example, g.V[0..9] to return the first 10 vertices.

26.2.16 TitanGraph status

status

Current graph life cycle status.

Parameters

Returns : str

Status of the graph

One of three statuses: Active, Deleted, Deleted_Final Active: available for use Deleted: has been scheduled for deletion Deleted_Final: backend files have been removed from disk.

26.2.17 TitanGraph vertex_sample

vertex_sample (*self*, *size*, *sample_type*, *seed=None*)

Make subgraph from vertex sampling.

Parameters **size** : int32

The number of vertices to sample from the graph.

sample_type : unicode

The type of vertex sample among: ['uniform', 'degree', 'degreedist'].

seed : int64 (default=None)

Random seed value.

Returns : dict

A new Graph object representing the vertex induced subgraph.

Create a vertex induced subgraph obtained by vertex sampling. Three types of vertex sampling are provided: ‘uniform’, ‘degree’, and ‘degreedist’. A ‘uniform’ vertex sample is obtained by sampling vertices uniformly at random. For ‘degree’ vertex sampling, each vertex is weighted by its out-degree. For ‘degreedist’ vertex sampling, each vertex is weighted by the total number of vertices that have the same out-degree as it. That is, the weight applied to each vertex for ‘degreedist’ vertex sampling is given by the out-degree histogram bin size.

class TitanGraph

Proxy to a graph in Titan, supports Gremlin query

Attributes

<code>name</code>	Set or get the name of the graph object.
<code>status</code>	Current graph life cycle status.

Methods

<code>__init__(self[, name, _info])</code>	Initialize the graph.
<code>__init__(self, entity)</code>	<Missing Doc>
<code>__init__(self, entity)</code>	<Missing Doc>
<code>annotate_degrees(self, output_property_name[, degree_option, ...])</code>	Make new graph with degrees.
<code>annotate_weighted_degrees(self, output_property_name[, ...])</code>	Calculates the weighted degree of each vertex with respect to an (optional) set of labels.
<code>clustering_coefficient(self[, output_property_name, ...])</code>	Coefficient of graph with respect to labels.
<code>copy(self[, name])</code>	Make a copy of the current graph.
<code>export_to_graph(self)</code>	Export from ta.TitanGraph to ta.Graph.
<code>graph_clustering(self, edge_distance)</code>	Performs graph clustering over an initial titan graph.
<code>graphx_connected_components(self, output_property)</code>	Implements the connected components computation on a graph by invoking graphx api.
<code>graphx_pagerank(self, output_property[, input_edge_labels, ...])</code>	Determine which vertices are the most important.
<code>graphx_triangle_count(self, output_property[, input_edge_labels])</code>	Number of triangles among vertices of current graph.
<code>ml.belief_propagation(self, prior_property, posterior_property)</code>	Classification on sparse data using Belief Propagation.
<code>query.gremlin(self, gremlin)</code>	Executes a Gremlin query.
<code>vertex_sample(self, size, sample_type[, seed])</code>	Make subgraph from vertex sampling.

`__init__(self, name=None)`

Initialize the graph.

Parameters `name` : (default=None)

26.3 *trustedanalytics* get_graph

get_graph (*identifier*)

Get handle to a graph object.

Parameters **identifier** : str | int
Name of the graph to get
Returns : Graph
graph object

26.4 *trustedanalytics* drop_graphs

drop_graphs (*items*)
Deletes the graph on the server.
Parameters **items** : [str | graph object | list [str | graph objects]]
Either the name of the graph object to delete or the graph object itself

26.5 *trustedanalytics* get_graph_names

get_graph_names ()
Retrieve names for all the graph objects on the server.
Returns : list
List of names

Global Methods

[get_graph](#)
[drop_graphs](#)
[get_graph_names](#)

MODELS

Classes

27.1 *Models LibsvmModel*

27.1.1 *LibsvmModel new*

__init__ (*self*, *name=None*)
[ALPHA] model:libsvm/new

Parameters **name** : unicode (default=None)

User supplied name.

Returns : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

27.1.2 *LibsvmModel name*

name

Set or get the name of the model object.

Parameters

Change or retrieve model object identification. Identification names must start with a letter and are limited to alphanumeric characters and the _ character.

Examples

```
>>> my_model.name

"csv_data"

>>> my_model.name = "cleaned_data"
>>> my_model.name

"cleaned_data"
```

27.1.3 *LibsvmModel* predict

predict (*self*, *frame*, *observation_columns=None*)

[ALPHA] New frame with new predicted label column.

Parameters **frame** : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame whose labels are to be predicted.

observation_columns : list (default=None)

Column(s) containing the observations whose labels are to be predicted. Default is the columns the *LibsvmModel* was trained on.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A new frame containing the original frame's columns and a column *predicted_label* containing the score calculated for each observation.

Predict the labels for a test frame and create a new frame revision with existing columns and a new predicted label's column.

27.1.4 *LibsvmModel* publish

publish (*self*)

[BETA] Creates a tar file that will used as input to the scoring engine

Parameters

Returns : dict

Returns the HDFS path to the tar file

27.1.5 *LibsvmModel* score

score (*self*, *vector*)

[ALPHA] Calculate the prediction label for a single observation.

Parameters **vector** : None

Returns : dict

Predicted label.

27.1.6 *LibsvmModel* test

test (*self*, *frame*, *label_column*, *observation_columns=None*)

[ALPHA] Predict test frame labels and return metrics.

Parameters **frame** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame whose labels are to be predicted.

label_column : unicode

Column containing the actual label for each observation.

observation_columns : list (default=None)

Column(s) containing the observations whose labels are to be predicted and tested.
Default is to test over the columns the LibsvmModel was trained on.

Returns : dict

Object Object with binary classification metrics. The data returned is composed of multiple components:

<object>.accuracy [double] The degree of correctness of the test frame labels.

<object>.confusion_matrix [table] A specific table layout that allows visualization of the performance of the test.

<object>.f_measure [double] A measure of a test's accuracy. It considers both the precision and the recall of the test to compute the score.

<object>.precision [double] The degree to which the correctness of the label is expressed.

<object>.recall [double] The fraction of relevant instances that are retrieved.

Predict the labels for a test frame and run classification metrics on predicted and target labels.

27.1.7 LibsvmModel train

train (*self, frame, label_column, observation_columns, svm_type=2, kernel_type=2, weight_label=None, weight=None, epsilon=0.001, degree=3, gamma=None, coef=0.0, nu=0.5, cache_size=100.0, shrinking=1, probability=0, nr_weight=1, c=1.0, p=0.1*)
[ALPHA] Train Lib Svm model based on another frame.

Parameters **frame** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame to train the model on.

label_column : unicode

Column name containing the label for each observation.

observation_columns : list

Column(s) containing the observations.

svm_type : int32 (default=2)

Set type of SVM. Default is one-class SVM.

0 – C-SVC 1 – nu-SVC 2 – one-class SVM 3 – epsilon-SVR 4 – nu-SVR

kernel_type : int32 (default=2)

Specifies the kernel type to be used in the algorithm. Default is RBF.

0 – linear: $u^T v$ 1 – polynomial: $(\gamma u^T v + \text{coef0})^{\text{degree}}$ 2 – radial basis
function: $\exp(-\gamma \|u - v\|^2)$ 3 – sigmoid: $\tanh(\gamma u^T v + \text{coef0})$

weight_label : list (default=None)

Default is (Array[Int](0))

weight : list (default=None)

Default is (Array[Double](0.0))

epsilon : float64 (default=0.001)

Set tolerance of termination criterion

degree : int32 (default=3)

Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

gamma : float64 (default=None)

Kernel coefficient for 'rbf', 'poly' and 'sigmoid'. Default is $1/n_{\text{features}}$.

coef : float64 (default=0.0)

Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

nu : float64 (default=0.5)

Set the parameter nu of nu-SVC, one-class SVM, and nu-SVR.

cache_size : float64 (default=100.0)

Specify the size of the kernel cache (in MB).

shrinking : int32 (default=1)

Whether to use the shrinking heuristic. Default is 1 (true).

probability : int32 (default=0)

Whether to enable probability estimates. Default is 0 (false).

nr_weight : int32 (default=1)

NR Weight

c : float64 (default=1.0)

Penalty parameter c of the error term.

p : float64 (default=0.1)

Set the epsilon in loss function of epsilon-SVR.

Returns : _Unit

Creating a lib Svm Model using the observation column and label column of the train frame.

class LibsvmModel

model:libsvm/new

Attributes

name	Set or get the name of the model object.
------	--

Methods

<code>__init__(self[, name, _info])</code>	[ALPHA] model:libsvm/new
<code>predict(self, frame[, observation_columns])</code>	[ALPHA] New frame with new predicted label column.
<code>publish(self)</code>	[BETA] Creates a tar file that will used as input to the scoring engine
<code>score(self, vector)</code>	[ALPHA] Calculate the prediction label for a single observation.
<code>test(self, frame, label_column[, observation_columns])</code>	[ALPHA] Predict test frame labels and return metrics.
<code>train(self, frame, label_column, observation_columns[, svm_type, ...])</code>	[ALPHA] Train Lib Svm model based on another frame.

`__init__(self, name=None)`
[ALPHA] model:libsvm/new

Parameters `name` : unicode (default=None)

User supplied name.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

27.2 Models RandomForestClassifierModel

27.2.1 RandomForestClassifierModel new

`__init__(self, name=None)`
Create a 'new' instance of random forest classifier model.

Parameters `name` : unicode (default=None)

User supplied name.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

27.2.2 RandomForestClassifierModel name

name

Set or get the name of the model object.

Parameters

Change or retrieve model object identification. Identification names must start with a letter and are limited to alphanumeric characters and the `_` character.

Examples

```
>>> my_model.name

"csv_data"

>>> my_model.name = "cleaned_data"
>>> my_model.name

"cleaned_data"
```

27.2.3 *RandomForestClassifierModel* predict

predict (*self*, *frame*, *observation_columns=None*)
[ALPHA] Predict the labels for the data points.

Parameters **frame** : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame whose labels are to be predicted. By default, predict is run on the same columns over which the model is trained.

observation_columns : list (default=None)

Column(s) containing the observations whose labels are to be predicted. By default, we predict the labels over columns the `RandomForestModel` was trained on.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

Frame A new frame consisting of the existing columns of the frame and a new column with predicted label for each observation.

27.2.4 *RandomForestClassifierModel* publish

publish (*self*)
[BETA] Creates a tar file that will be used as input to the scoring engine

Parameters

Returns : dict

Returns the HDFS path to the tar file

Creates a tar file with the trained Random Forest Classifier Model The tar file is used as input to the scoring engine to predict the class of an observation.

27.2.5 *RandomForestClassifierModel* test

test (*self*, *frame*, *label_column*, *observation_columns=None*)
[ALPHA] Predict test frame labels and return metrics.

Parameters **frame** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

The frame whose labels are to be predicted

label_column : unicode

Column containing the true labels of the observations

observation_columns : list (default=None)

Column(s) containing the observations whose labels are to be predicted. By default, we predict the labels over columns the RandomForest was trained on.

Returns : dict

object

An object with classification metrics. The data returned is composed of multiple components:

<object>.accuracy : double <object>.confusion_matrix : table <object>.f_measure :
double <object>.precision : double <object>.recall : double

Predict the labels for a test frame and run classification metrics on predicted and target labels.

27.2.6 RandomForestClassifierModel train

train (*self*, *frame*, *label_column*, *observation_columns*, *num_classes*=2, *num_trees*=1, *impurity*='gini',
max_depth=4, *max_bins*=100, *seed*=-842726101, *categorical_features_info*=None, *feature_subset_category*=None)
[ALPHA] Build Random Forests Classifier model.

Parameters **frame** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame to train the model on.

label_column : unicode

Column name containing the label for each observation.

observation_columns : list

Column(s) containing the observations.

num_classes : int32 (default=2)

Number of classes for classification

num_trees : int32 (default=1)

Number of trees in the random forest

impurity : unicode (default=gini)

Criterion used for information gain calculation. Supported values “gini” or “entropy”

max_depth : int32 (default=4)

Maximum depth of the tree

max_bins : int32 (default=100)

Maximum number of bins used for splitting features

seed : int32 (default=-842726101)

Random seed for bootstrapping and choosing feature subsets

categorical_features_info : None (default=None)

feature_subset_category : unicode (default=None)

Number of features to consider for splits at each node. Supported values
“auto”, “all”, “sqrt”, “log2”, “onethird”

Returns : dict

Creating a Random Forests Classifier Model using the observation columns and label column.

class RandomForestClassifierModel

Create a ‘new’ instance of random forest classifier model.

Attributes

name	Set or get the name of the model object.
-------------	--

Methods

__init__ (self[, name, _info])	Create a ‘new’ instance of random forest classifier model.
predict (self, frame[, observation_columns])	[ALPHA] Predict the labels for the data points.
publish (self)	[BETA] Creates a tar file that will be used as input to the scoring engine
test (self, frame, label_column[, observation_columns])	[ALPHA] Predict test frame labels and return metrics.
train (self, frame, label_column, observation_columns[, num_classes, ...])	[ALPHA] Build Random Forests Classifier model.

__init__(self, name=None)

Create a ‘new’ instance of random forest classifier model.

Parameters **name** : unicode (default=None)

User supplied name.

Returns : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

27.3 Models PrincipalComponentsModel

27.3.1 PrincipalComponentsModel new

__init__(self, name=None)

Create a ‘new’ instance of principal component model.

Parameters **name** : unicode (default=None)

User supplied name.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

27.3.2 *PrincipalComponentsModel* name

name

Set or get the name of the model object.

Parameters

Change or retrieve model object identification. Identification names must start with a letter and are limited to alphanumeric characters and the `_` character.

Examples

```
>>> my_model.name

"csv_data"

>>> my_model.name = "cleaned_data"
>>> my_model.name

"cleaned_data"
```

27.3.3 *PrincipalComponentsModel* predict

predict (*self*, *frame*, *mean_centered=True*, *t_squared_index=False*, *observation_columns=None*, *c=None*,
name=None)
[ALPHA] Predict using principal components model.

Parameters **frame** : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

Frame whose principal components are to be computed.

mean_centered : bool (default=True)

Option to mean center the columns. Default is true

t_squared_index : bool (default=False)

Indicator for whether the t-square index is to be computed. Default is false.

observation_columns : list (default=None)

List of observation column name(s) to be used for prediction. Default is the list of column name(s) used to train the model.

c : int32 (default=None)

The number of principal components to be predicted. Default is the count used to train the model.

name : unicode (default=None)

The name of the output frame generated by predict.

Returns : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame with existing columns and 'c' additional columns containing the projections of V on the the frame and an additional column storing the t-square-index value if requested

Predicting on a dataframe's columns using a PrincipalComponents Model.

27.3.4 *PrincipalComponentsModel* publish

publish (*self*)

[BETA] Creates a tar file that will be used as input to the scoring engine

Parameters

Returns : dict

Returns the HDFS path to the tar file

Creates a tar file with the trained Principal Components Model. The tar file is used as input to the scoring engine to compute the principal components and t-squared index of the observation.

27.3.5 *PrincipalComponentsModel* train

train (*self*, *frame*, *observation_columns*, *mean_centered=True*, *k=None*)

Build principal components model.

Parameters **frame** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame to train the model on.

observation_columns : list

List of column(s) containing the observations.

mean_centered : bool (default=True)

Option to mean center the columns

k : int32 (default=None)

Principal component count. Default is the number of observation columns

Returns : dict

Creating a PrincipalComponents Model using the observation columns.

class PrincipalComponentsModel

Create a 'new' instance of principal component model.

Attributes

name	Set or get the name of the model object.
-------------	--

Methods

<code>__init__(self[, name, _info])</code>	Create a 'new' instance of principal component model.
<code>predict(self, frame[, mean_centered, t_squared_index, observation_columns, ...])</code>	[ALPHA] Predict using principal components model.
<code>publish(self)</code>	[BETA] Creates a tar file that will be used as input to the scoring engine
<code>train(self, frame, observation_columns[, mean_centered, k])</code>	Build principal components model.

`__init__(self, name=None)`

Create a 'new' instance of principal component model.

Parameters **name** : unicode (default=None)

User supplied name.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

27.4 Models CollaborativeFilteringModel

27.4.1 CollaborativeFilteringModel new

`__init__(self, name=None)`

Collaborative filtering recommend model.

Parameters **name** : unicode (default=None)

User supplied name.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

Collaborative Filtering

Collaborative filtering is a technique that is widely used in recommendation systems to suggest items (for example, products, movies, articles) to potential users based on historical records of items that users have purchased, rated, or viewed. The Trusted Analytics provides implementations of collaborative filtering with either Alternating Least Squares (ALS) or Conjugate Gradient Descent (CGD) optimization methods.

Both methods optimize the cost function found in Y. Koren, [Factorization Meets the Neighborhood: a Multifaceted Collaborative Filtering Model](http://public.research.att.com/~volinsky/netflix/kdd08koren.pdf)¹ in ACM KDD 2008. For more information on optimizing using ALS see, Y. Zhou, D. Wilkinson, R. Schreiber and R. Pan, [Large-Scale Parallel Collaborative Filtering for the Netflix Prize](http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.173.2797)², 2008.

¹<http://public.research.att.com/~volinsky/netflix/kdd08koren.pdf>

²<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.173.2797>

CGD provides a faster, more approximate optimization of the cost function and should be used when memory is a constraint.

A typical representation of the preference matrix P in Giraph is a bi-partite graph, where nodes at the left side represent a list of users and nodes at the right side represent a set of items (for example, movies), and edges encode the rating a user provided to an item. To support training, validation and test, a common practice in machine learning, each edge is also annotated by “TR”, “VA” or “TE”.

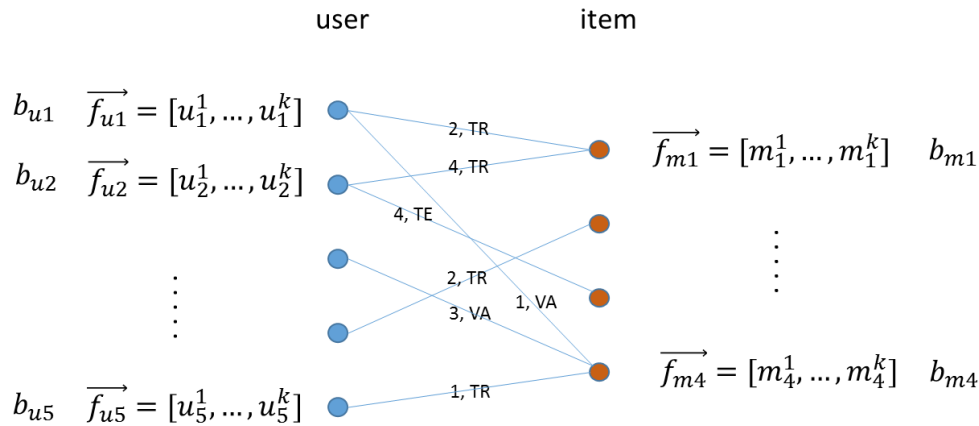


Fig. 27.1: A typical representation of the preference matrix P

Each node in the graph will be associated with a vector \vec{f}_x of length k , where k is the feature dimension specified by the user, and a bias term b_x . The predictions for item m_j , from user u_i are given by dot product of the feature vector and the user vector, plus the item and user bias terms: `/home/work/atk/engine-plugins/giraph-plugins/src/main/scala/org/trustedanalytics/atk/giraph/plugins/model/cf/CollaborativeFilteringNewPlugin.scala`

$$r_{ij} = \vec{f}_{ui} \cdot \vec{f}_{mj} + b_{ui} + b_{mj}$$

The parameters of the above equation are chosen to minimize the regularized mean squared error between known and predicted ratings:

$$cost = \frac{\sum error^2}{n} + \lambda * (bias^2 + \sum f_k^2)$$

How this optimization is accomplished depends on whether the user uses the ALS or CGD functions respectively. It is recommended that the ALS method be used to solve collaborative filtering problems. The CGD method uses less memory than ALS, but it returns an approximate solution to the objective function and should only be used in cases when memory required for ALS is prohibitively high.

Using ALS Optimization to Solve the Collaborative Filtering Problem

ALS optimizes the vector \vec{f}_* and the bias b_* alternatively between user profiles using least squares on users and items. On the first iteration, the first feature of each item is set to its average rating, while the others are set to small random numbers. The algorithm then treats the m 's as constant and optimizes u_i^1, \dots, u_i^k for each user, i . For an individual user, this is a simple ordinary least squares optimization over the items that user has ranked. Next, the algorithm takes the u 's as constant and optimizes the m_j^1, \dots, m_j^k for each item, j . This is again an ordinary least squares optimization predicting the user rating of person that has ranked item j .

At each step, the bias is computed for either items or users and the objective function, shown below, is evaluated. The bias term for an item or user, computed for use in the next iteration is given by:

$$b = \frac{\sum error}{(1 + \lambda) * n}$$

The optimization is said to converge if the change in the objective function is less than the `convergence_threshold` parameter or the algorithm hits the maximum number of *supersteps*.

$$cost = \frac{\sum error^2}{n} + \lambda * \left(bias^2 + \sum f_k^2 \right)$$

Note that the equations above omit user and item subscripts for generality. The l_2 regularization term, λ , tries to avoid overfitting by penalizing the magnitudes of the parameters, and λ is a tradeoff parameter that balances the two terms and is usually determined by cross validation (CV).

After the parameters \vec{f}_* and b_* are determined, given an item m_j the rating from user u_i can be predicted by the simple linear model:

$$r_{ij} = \vec{f}_{ui} \cdot \vec{f}_{mj} + b_{ui} + b_{mj}$$

Matrix Factorization based on Conjugate Gradient Descent (CGD)

This is the Conjugate Gradient Descent (CGD) with Bias for collaborative filtering algorithm. Our implementation is based on the paper:

Y. Koren. Factorization Meets the Neighborhood: a Multifaceted Collaborative Filtering Model. In ACM KDD 2008. (Equation 5) <http://public.research.att.com/~volinsky/netflix/kdd08koren.pdf>

This algorithm for collaborative filtering is used in *recommendation systems* to suggest items (products, movies, articles, and so on) to potential users based on historical records of items that all users have purchased, rated, or viewed. The records are usually organized as a preference matrix P , which is a sparse matrix holding the preferences (such as, ratings) given by users to items. Similar to ALS, CGD falls in the category of matrix factorization/latent factor model that infers user profiles and item profiles in low-dimension space, such that the original matrix P can be approximated by a linear model.

This factorization method uses the conjugate gradient method for its optimization subroutine. For more on conjugate gradient descent in general, see: http://en.wikipedia.org/wiki/Conjugate_gradient_method.

The Mathematics of Matrix Factorization via CGD

Matrix factorization by conjugate gradient descent produces ratings by using the (limited) space of observed rankings to infer a user-factors vector p_u for each user u , and an item-factors vector q_i for each item i , and then producing a ranking by user u of item i by the dot-product $b_{ui} + p_u^T q_i$ where b_{ui} is a baseline ranking calculated as $b_{ui} = \mu + b_u + b_i$.

The optimum model is chosen to minimum the following sum, which penalizes square distance of the prediction from observed rankings and complexity of the model (through the regularization term):

$$\sum_{(u,i) \in \mathcal{K}} (r_{ui} - \mu - b_u - b_i - p_u^T q_i)^2 + \lambda_3 (||p_u||^2 + ||q_i||^2 + b_u^2 + b_i^2)$$

Where:

- r_{ui} — Observed ranking of item i by user u
- \mathcal{K} — Set of pairs (u, i) for each observed ranking of item i by user u
- μ — The average rating over all ratings of all items by all users.
- b_u — How much user u 's average rating differs from μ .
- b_i — How much item i 's average rating differs from μ
- p_u — User-factors vector.
- q_i — Item-factors vector.
- λ_3 — A regularization parameter specified by the user.

This optimization problem is solved by the conjugate gradient descent method. Indeed, this difference in how the optimization problem is solved is the primary difference between matrix factorization by CGD and matrix factorization by ALS.

Comparison between CGD and ALS

Both CGD and ALS provide recommendation systems based on matrix factorization; the difference is that CGD employs the conjugate gradient descent instead of least squares for its optimization phase. In particular, they share the same bipartite graph representation and the same cost function.

- ALS finds a better solution faster - when it can run on the cluster it is given.
- CGD has slighter memory requirements and can run on datasets that can overwhelm the ALS-based solution.

When feasible, ALS is a preferred solver over CGD, while CGD is recommended only when the application requires so much memory that it might be beyond the capacity of the system. CGD has a smaller memory requirement, but has a slower rate of convergence and can provide a rougher estimate of the solution than the more computationally intensive ALS.

The reason for this is that ALS solves the optimization problem by a least squares that requires inverting a matrix. Therefore, it requires more memory and computational effort. But ALS, a 2nd-order optimization method, enjoys higher convergence rate and is potentially more accurate in parameter estimation.

On the otherhand, CGD is a 1.5th-order optimization method that approximates the Hessian of the cost function from the previous gradient information through N consecutive CGD updates. This is very important in cases where the solution has thousands or even millions of components.

Usage

The matrix factorization by CGD procedure takes a property graph, encoding a bipartite user-item ranking network, selects a subset of the edges to be considered (via a selection of edge labels), takes initial ratings from specified edge property values, and then writes each user-factors vector to its user vertex in a specified vertex property name and each item-factors vector to its item vertex in the specified vertex property name.

27.4.2 CollaborativeFilteringModel name

name

Set or get the name of the model object.

Parameters

Change or retrieve model object identification. Identification names must start with a letter and are limited to alphanumeric characters and the `_` character.

Examples

```
>>> my_model.name

"csv_data"

>>> my_model.name = "cleaned_data"
>>> my_model.name

"cleaned_data"
```

27.4.3 CollaborativeFilteringModel recommend

recommend (*self*, *name*, *top_k*)

[BETA] Collaborative filtering (als/cgd) model

Parameters **name** : unicode

An entity name from the first column of the input frame

top_k : int32

positive integer representing the top recommendations for the name

Returns : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

see collaborative filtering train for more information

see collaborative filtering train for more information

27.4.4 CollaborativeFilteringModel train

train (*self*, *frame*, *user_col_name*, *item_col_name*, *rating_col_name*, *evaluation_function=None*,
num_factors=None, *max_iterations=None*, *convergence_threshold=None*, *regularization=None*,
bias_on=None, *min_value=None*, *max_value=None*, *learning_curve_interval=None*,
cgd_iterations=None)

Collaborative filtering (als/cgd) model

Parameters **frame** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

user_col_name : unicode

Name of the user column from input data

item_col_name : unicode

Name of the item column from input data

rating_col_name : unicode

Name of the rating column from input data

evaluation_function : unicode (default=None)

als/cgd

num_factors : int32 (default=None)

Size of the desired factors (default is 3)

max_iterations : int32 (default=None)

Max number of iterations for Giraph

convergence_threshold : float64 (default=None)

float value between 0 .. 1

regularization : float32 (default=None)

float value between 0 .. 1

bias_on : bool (default=None)

bias on/off switch

min_value : float32 (default=None)

minimum edge weight value

max_value : float32 (default=None)

minimum edge weight value

learning_curve_interval : int32 (default=None)

iteration interval to output learning curve

cgd_iterations : int32 (default=None)

custom argument for cgd learning curve output interval (default: every iteration)

Returns : dict

Execution result summary for Giraph

class CollaborativeFilteringModel

Collaborative filtering recommend model.

Collaborative Filtering

Collaborative filtering is a technique that is widely used in recommendation systems to suggest items (for example, products, movies, articles) to potential users based on historical records of items that users have purchased, rated, or viewed. The Trusted Analytics provides implementations of collaborative filtering with either Alternating Least Squares (ALS) or Conjugate Gradient Descent (CGD) optimization methods.

Both methods optimize the cost function found in Y. Koren, [Factorization Meets the Neighborhood: a Multifaceted Collaborative Filtering Model](#)³ in ACM KDD 2008. For more information on optimizing using ALS see, Y. Zhou, D. Wilkinson, R. Schreiber and R. Pan, [Large-Scale Parallel Collaborative Filtering for the Netflix Prize](#)⁴, 2008.

CGD provides a faster, more approximate optimization of the cost function and should be used when memory is a constraint.

A typical representation of the preference matrix P in Giraph is a bi-partite graph, where nodes at the left side represent a list of users and nodes at the right side represent a set of items (for example, movies), and edges encode the rating a user provided to an item. To support training, validation and test, a common practice in machine learning, each edge is also annotated by “TR”, “VA” or “TE”.

Each node in the graph will be associated with a vector \vec{f}_x of length k , where k is the feature dimension specified by the user, and a bias term b_x . The predictions for item m_j , from user u_i are given by dot product of the feature vector and the user vector, plus the item and user bias terms: `/home/work/atk/engine-plugins/giraph-plugins/src/main/scala/org/trustedanalytics/atk/giraph/plugins/model/cf/CollaborativeFilteringNewPlugin.scala`

$$r_{ij} = \vec{f}_{ui} \cdot \vec{f}_{mj} + b_{ui} + b_{mj}$$

The parameters of the above equation are chosen to minimize the regularized mean squared error between known and predicted ratings:

$$cost = \frac{\sum error^2}{n} + \lambda * (bias^2 + \sum f_k^2)$$

How this optimization is accomplished depends on whether the use uses the ALS or CGD functions respectively. It is recommended that the ALS method be used to solve collaborative filtering problems. The

³<http://public.research.att.com/~volinsky/netflix/kdd08koren.pdf>

⁴<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.173.2797>

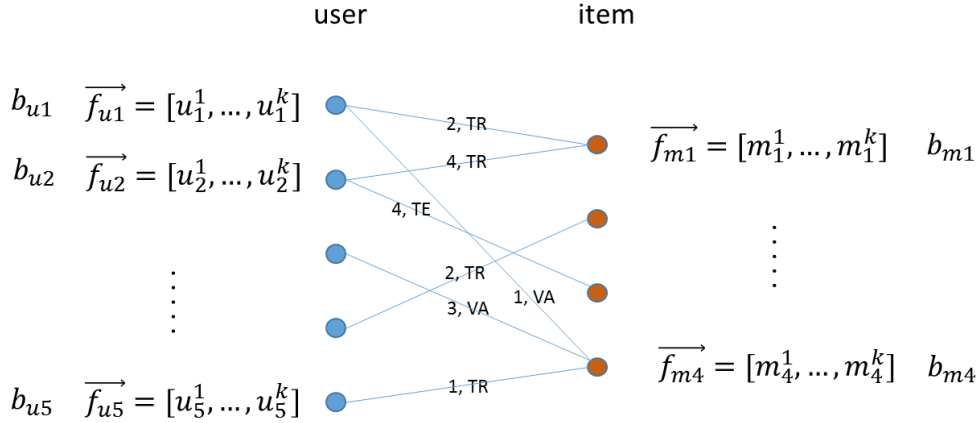


Fig. 27.2: A typical representation of the preference matrix P

CGD method uses less memory than ALS, but it returns an approximate solution to the objective function and should only be used in cases when memory required for ALS is prohibitively high.

Using ALS Optimization to Solve the Collaborative Filtering Problem

ALS optimizes the vector \vec{f}_* and the bias b_* alternatively between user profiles using least squares on users and items. On the first iteration, the first feature of each item is set to its average rating, while the others are set to small random numbers. The algorithm then treats the m 's as constant and optimizes u_i^1, \dots, u_i^k for each user, i . For an individual user, this is a simple ordinary least squares optimization over the items that user has ranked. Next, the algorithm takes the u 's as constant and optimizes the m_j^1, \dots, m_j^k for each item, j . This is again an ordinary least squares optimization predicting the user rating of person that has ranked item j .

At each step, the bias is computed for either items or users and the objective function, shown below, is evaluated. The bias term for an item or user, computed for use in the next iteration is given by:

$$b = \frac{\sum error}{(1 + \lambda) * n}$$

The optimization is said to converge if the change in the objective function is less than the `convergence_threshold` parameter or the algorithm hits the maximum number of *supersteps*.

$$cost = \frac{\sum error^2}{n} + \lambda * \left(bias^2 + \sum f_k^2 \right)$$

Note that the equations above omit user and item subscripts for generality. The l_2 regularization term, lambda, tries to avoid overfitting by penalizing the magnitudes of the parameters, and λ is a tradeoff parameter that balances the two terms and is usually determined by cross validation (CV).

After the parameters \vec{f}_* and b_* are determined, given an item m_j the rating from user u_i can be predicted by the simple linear model:

$$r_{ij} = \vec{f}_{ui} \cdot \vec{f}_{mj} + b_{ui} + b_{mj}$$

Matrix Factorization based on Conjugate Gradient Descent (CGD)

This is the Conjugate Gradient Descent (CGD) with Bias for collaborative filtering algorithm. Our implementation is based on the paper:

Y. Koren. Factorization Meets the Neighborhood: a Multifaceted Collaborative Filtering Model. In ACM KDD 2008. (Equation 5) <http://public.research.att.com/~volinsky/netflix/kdd08koren.pdf>

This algorithm for collaborative filtering is used in *recommendation systems* to suggest items (products, movies, articles, and so on) to potential users based on historical records of items that all users have purchased, rated, or viewed. The records are usually organized as a preference matrix P , which is a sparse matrix holding the preferences (such as, ratings) given by users to items. Similar to ALS, CGD falls in the category of matrix factorization/latent factor model that infers user profiles and item profiles in low-dimension space, such that the original matrix P can be approximated by a linear model.

This factorization method uses the conjugate gradient method for its optimization subroutine. For more on conjugate gradient descent in general, see: http://en.wikipedia.org/wiki/Conjugate_gradient_method.

The Mathematics of Matrix Factorization via CGD

Matrix factorization by conjugate gradient descent produces ratings by using the (limited) space of observed rankings to infer a user-factors vector p_u for each user u , and an item-factors vector q_i for each item i , and then producing a ranking by user u of item i by the dot-product $b_{ui} + p_u^T q_i$ where b_{ui} is a baseline ranking calculated as $b_{ui} = \mu + b_u + b_i$.

The optimum model is chosen to minimum the following sum, which penalizes square distance of the prediction from observed rankings and complexity of the model (through the regularization term):

$$\sum_{(u,i) \in \mathcal{K}} (r_{ui} - \mu - b_u - b_i - p_u^T q_i)^2 + \lambda_3 (\|p_u\|^2 + \|q_i\|^2 + b_u^2 + b_i^2)$$

Where:

- r_{ui} — Observed ranking of item i by user u
- \mathcal{K} — Set of pairs (u, i) for each observed ranking of item i by user u
- μ — The average rating over all ratings of all items by all users.
- b_u — How much user u 's average rating differs from μ .
- b_i — How much item i 's average rating differs from μ
- p_u — User-factors vector.
- q_i — Item-factors vector.
- λ_3 — A regularization parameter specified by the user.

This optimization problem is solved by the conjugate gradient descent method. Indeed, this difference in how the optimization problem is solved is the primary difference between matrix factorization by CGD and matrix factorization by ALS.

Comparison between CGD and ALS

Both CGD and ALS provide recommendation systems based on matrix factorization; the difference is that CGD employs the conjugate gradient descent instead of least squares for its optimization phase. In particular, they share the same bipartite graph representation and the same cost function.

- ALS finds a better solution faster - when it can run on the cluster it is given.
- CGD has slighter memory requirements and can run on datasets that can overwhelm the ALS-based solution.

When feasible, ALS is a preferred solver over CGD, while CGD is recommended only when the application requires so much memory that it might be beyond the capacity of the system. CGD has a smaller memory requirement, but has a slower rate of convergence and can provide a rougher estimate of the solution than the more computationally intensive ALS.

The reason for this is that ALS solves the optimization problem by a least squares that requires inverting a matrix. Therefore, it requires more memory and computational effort. But ALS, a 2nd-order optimization method, enjoys higher convergence rate and is potentially more accurate in parameter estimation.

On the otherhand, CGD is a 1.5th-order optimization method that approximates the Hessian of the cost function from the previous gradient information through N consecutive CGD updates. This is very important in cases where the solution has thousands or even millions of components.

Usage

The matrix factorization by CGD procedure takes a property graph, encoding a biparite user-item ranking network, selects a subset of the edges to be considered (via a selection of edge labels), takes initial ratings from specified edge property values, and then writes each user-factors vector to its user vertex in a specified vertex property name and each item-factors vector to its item vertex in the specified vertex property name.

Attributes

<code>name</code>	Set or get the name of the model object.
-------------------	--

Methods

<code>__init__(self[, name, _info])</code>	Collaborative filtering recommend model.
<code>recommend(self, name, top_k)</code>	[BETA] Collaborative filtering (als/cgd) model
<code>train(self, frame, user_col_name, item_col_name, rating_col_name[, ...])</code>	Collaborative filtering (als/cgd) model

`__init__(self, name=None)`

Collaborative filtering recommend model.

Parameters `name` : unicode (default=None)

User supplied name.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

Collaborative Filtering

Collaborative filtering is a technique that is widely used in recommendation systems to suggest items (for example, products, movies, articles) to potential users based on historical records of items that users have purchased, rated, or viewed. The Trusted Analytics provides implementations of collaborative filtering with either Alternating Least Squares (ALS) or Conjugate Gradient Descent (CGD) optimization methods.

Both methods optimize the cost function found in Y. Koren, [Factorization Meets the Neighborhood: a Multifaceted Collaborative Filtering Model](http://public.research.att.com/~volinsky/netflix/kdd08koren.pdf)⁵ in ACM KDD 2008. For more information on optimizing using ALS see, Y. Zhou, D. Wilkinson, R. Schreiber and R. Pan, [Large-Scale Parallel Collaborative Filtering for the Netflix Prize](http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.173.2797)⁶, 2008.

CGD provides a faster, more approximate optimization of the cost function and should be used when memory is a constraint.

A typical representation of the preference matrix P in Giraph is a bi-partite graph, where nodes at the left side represent a list of users and nodes at the right side represent a set of items (for example, movies), and edges encode the rating a user provided to an item. To support training, validation and test, a common practice in machine learning, each edge is also annotated by “TR”, “VA” or “TE”.

⁵<http://public.research.att.com/~volinsky/netflix/kdd08koren.pdf>

⁶<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.173.2797>

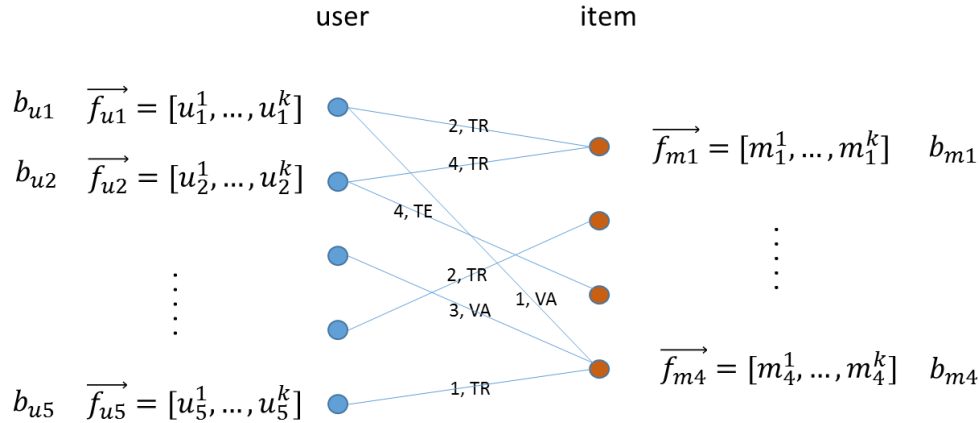


Fig. 27.3: A typical representation of the preference matrix P

Each node in the graph will be associated with a vector \vec{f}_x of length k , where k is the feature dimension specified by the user, and a bias term b_x . The predictions for item m_j , from user u_i are given by dot product of the feature vector and the user vector, plus the item and user bias terms: `/home/work/atk/engine-plugins/giraph-plugins/src/main/scala/org/trustedanalytics/atk/giraph/plugins/model/cf/CollaborativeFilteringNewPlugin.scala`

$$r_{ij} = \vec{f}_{ui} \cdot \vec{f}_{mj} + b_{ui} + b_{mj}$$

The parameters of the above equation are chosen to minimize the regularized mean squared error between known and predicted ratings:

$$cost = \frac{\sum error^2}{n} + \lambda * (bias^2 + \sum f_k^2)$$

How this optimization is accomplished depends on whether the user uses the ALS or CGD functions respectively. It is recommended that the ALS method be used to solve collaborative filtering problems. The CGD method uses less memory than ALS, but it returns an approximate solution to the objective function and should only be used in cases when memory required for ALS is prohibitively high.

Using ALS Optimization to Solve the Collaborative Filtering Problem

ALS optimizes the vector \vec{f}_* and the bias b_* alternatively between user profiles using least squares on users and items. On the first iteration, the first feature of each item is set to its average rating, while the others are set to small random numbers. The algorithm then treats the m 's as constant and optimizes u_i^1, \dots, u_i^k for each user, i . For an individual user, this is a simple ordinary least squares optimization over the items that user has ranked. Next, the algorithm takes the u 's as constant and optimizes the m_j^1, \dots, m_j^k for each item, j . This is again an ordinary least squares optimization predicting the user rating of person that has ranked item j .

At each step, the bias is computed for either items or users and the objective function, shown below, is evaluated. The bias term for an item or user, computed for use in the next iteration is given by:

$$b = \frac{\sum error}{(1 + \lambda) * n}$$

The optimization is said to converge if the change in the objective function is less than the `convergence_threshold` parameter or the algorithm hits the maximum number of *supersteps*.

$$cost = \frac{\sum error^2}{n} + \lambda * (bias^2 + \sum f_k^2)$$

Note that the equations above omit user and item subscripts for generality. The l_2 regularization term, lambda, tries to avoid overfitting by penalizing the magnitudes of the parameters, and λ is a tradeoff parameter that balances the two terms and is usually determined by cross validation (CV).

After the parameters \vec{f}_* and b_* are determined, given an item m_j the rating from user u_i can be predicted by the simple linear model:

$$r_{ij} = \vec{f}_{ui} \cdot \vec{f}_{mj} + b_{ui} + b_{mj}$$

Matrix Factorization based on Conjugate Gradient Descent (CGD)

This is the Conjugate Gradient Descent (CGD) with Bias for collaborative filtering algorithm. Our implementation is based on the paper:

Y. Koren. Factorization Meets the Neighborhood: a Multifaceted Collaborative Filtering Model. In ACM KDD 2008. (Equation 5) <http://public.research.att.com/~volinsky/netflix/kdd08koren.pdf>

This algorithm for collaborative filtering is used in *recommendation systems* to suggest items (products, movies, articles, and so on) to potential users based on historical records of items that all users have purchased, rated, or viewed. The records are usually organized as a preference matrix P , which is a sparse matrix holding the preferences (such as, ratings) given by users to items. Similar to ALS, CGD falls in the category of matrix factorization/latent factor model that infers user profiles and item profiles in low-dimension space, such that the original matrix P can be approximated by a linear model.

This factorization method uses the conjugate gradient method for its optimization subroutine. For more on conjugate gradient descent in general, see: http://en.wikipedia.org/wiki/Conjugate_gradient_method.

The Mathematics of Matrix Factorization via CGD

Matrix factorization by conjugate gradient descent produces ratings by using the (limited) space of observed rankings to infer a user-factors vector p_u for each user u , and an item-factors vector q_i for each item i , and then producing a ranking by user u of item i by the dot-product $b_{ui} + p_u^T q_i$ where b_{ui} is a baseline ranking calculated as $b_{ui} = \mu + b_u + b_i$.

The optimum model is chosen to minimum the following sum, which penalizes square distance of the prediction from observed rankings and complexity of the model (through the regularization term):

$$\sum_{(u,i) \in \mathcal{K}} (r_{ui} - \mu - b_u - b_i - p_u^T q_i)^2 + \lambda_3 (\|p_u\|^2 + \|q_i\|^2 + b_u^2 + b_i^2)$$

Where:

- r_{ui} — Observed ranking of item i by user u
- \mathcal{K} — Set of pairs (u, i) for each observed ranking of item i by user u
- μ — The average rating over all ratings of all items by all users.
- b_u — How much user u 's average rating differs from μ .
- b_i — How much item i 's average rating differs from μ
- p_u — User-factors vector.
- q_i — Item-factors vector.
- λ_3 — A regularization parameter specified by the user.

This optimization problem is solved by the conjugate gradient descent method. Indeed, this difference in how the optimization problem is solved is the primary difference between matrix factorization by CGD and matrix factorization by ALS.

Comparison between CGD and ALS

Both CGD and ALS provide recommendation systems based on matrix factorization; the difference is that CGD employs the conjugate gradient descent instead of least squares for its optimization phase. In particular, they share the same bipartite graph representation and the same cost function.

- ALS finds a better solution faster - when it can run on the cluster it is given.
- CGD has slighter memory requirements and can run on datasets that can overwhelm the ALS-based solution.

When feasible, ALS is a preferred solver over CGD, while CGD is recommended only when the application requires so much memory that it might be beyond the capacity of the system. CGD has a smaller memory requirement, but has a slower rate of convergence and can provide a rougher estimate of the solution than the more computationally intensive ALS.

The reason for this is that ALS solves the optimization problem by a least squares that requires inverting a matrix. Therefore, it requires more memory and computational effort. But ALS, a 2nd-order optimization method, enjoys higher convergence rate and is potentially more accurate in parameter estimation.

On the otherhand, CGD is a 1.5th-order optimization method that approximates the Hessian of the cost function from the previous gradient information through N consecutive CGD updates. This is very important in cases where the solution has thousands or even millions of components.

Usage

The matrix factorization by CGD procedure takes a property graph, encoding a biparite user-item ranking network, selects a subset of the edges to be considered (via a selection of edge labels), takes initial ratings from specified edge property values, and then writes each user-factors vector to its user vertex in a specified vertex property name and each item-factors vector to its item vertex in the specified vertex property name.

27.5 Models KMeansModel

27.5.1 KMeansModel new

`__init__(self, name=None)`
create a new model

Parameters `name` : unicode (default=None)
name for the model

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

27.5.2 KMeansModel name

name
Set or get the name of the model object.

Parameters

Change or retrieve model object identification. Identification names must start with a letter and are limited to alphanumeric characters and the `_` character.

Examples

```
>>> my_model.name

"csv_data"

>>> my_model.name = "cleaned_data"
>>> my_model.name

"cleaned_data"
```

27.5.3 *KMeansModel* predict

predict (*self*, *frame*, *observation_columns=None*)

[BETA] Predict the cluster assignments for the data points.

Parameters **frame** : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame whose labels are to be predicted. By default, predict is run on the same columns over which the model is trained.

observation_columns : list (default=None)

Column(s) containing the observations whose clusters are to be predicted. By default, we predict the clusters over columns the *KMeansModel* was trained on. The columns are scaled using the same values used when training the model.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

Frame A new frame consisting of the existing columns of the frame and new columns. The data returned is composed of multiple components:

‘k’ columns [double] Containing squared distance of each point to every cluster center.

predicted_cluster [int] Integer containing the cluster assignment.

27.5.4 *KMeansModel* publish

publish (*self*)

[BETA] Creates a tar file that will used as input to the scoring engine

Parameters

Returns : dict

Returns the HDFS path to the tar file

27.5.5 *KMeansModel* train

train (*self*, *frame*, *observation_columns*, *column_scalings*, *k=None*, *max_iterations=None*, *epsilon=None*, *initialization_mode=None*)
[BETA] Creates KMeans Model from train frame.

Parameters **frame** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame to train the model on.

observation_columns : list

Columns containing the observations.

column_scalings : list

Column scalings for each of the observation columns. The scaling value is multiplied by the corresponding value in the observation column.

k : int32 (default=None)

Desired number of clusters. Default is 2.

max_iterations : int32 (default=None)

Number of iterations for which the algorithm should run. Default is 20.

epsilon : float64 (default=None)

Distance threshold within which we consider k-means to have converged. Default is 1e-4.

initialization_mode : unicode (default=None)

The initialization technique for the algorithm. It could be either “random” or “k-meansll”. Default is “k-meansll”.

Returns : dict

dict Results. The data returned is composed of multiple components:

cluster_size [dict] Cluster size

ClusterId [int] Number of elements in the cluster ‘ClusterId’.

within_set_sum_of_squared_error [double] The set of sum of squared error for the model.

Upon training the ‘k’ cluster centers are computed.

class KMeansModel

create a new model

Attributes

name	Set or get the name of the model object.
------	--

Methods

<code>__init__(self[, name, _info])</code>	create a new model
<code>predict(self, frame[, observation_columns])</code>	[BETA] Predict the cluster assignments for the data points.
<code>publish(self)</code>	[BETA] Creates a tar file that will used as input to the scoring engine
<code>train(self, frame, observation_columns, column_scalings[, k, max_iterations, ...])</code>	[BETA] Creates KMeans Model from train frame.

`__init__(self, name=None)`
create a new model

Parameters **name** : unicode (default=None)

name for the model

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

27.6 Models SvmModel

27.6.1 SvmModel new

`__init__(self, name=None)`
[ALPHA] create a new model

Parameters **name** : unicode (default=None)

User supplied name.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

27.6.2 SvmModel name

name

Set or get the name of the model object.

Parameters

Change or retrieve model object identification. Identification names must start with a letter and are limited to alphanumeric characters and the `_` character.

Examples

```
>>> my_model.name

"csv_data"

>>> my_model.name = "cleaned_data"
>>> my_model.name

"cleaned_data"
```

27.6.3 *SvmModel* predict

predict (*self*, *frame*, *observation_columns=None*)

[ALPHA] Make new frame with additional column for predicted label.

Parameters **frame** : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame whose labels are to be predicted. By default, predict is run on the same columns over which the model is trained.

observation_columns : list (default=None)

Column(s) containing the observations whose labels are to be predicted. By default, we predict the labels over columns the `LogisticRegressionModel` was trained on.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame containing the original frame's columns and a column with the predicted label

Predict the labels for a test frame and create a new frame revision with existing columns and a new predicted label's column.

27.6.4 *SvmModel* test

test (*self*, *frame*, *label_column*, *observation_columns=None*)

[ALPHA] Predict test frame labels and return metrics.

Parameters **frame** : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

frame whose labels are to be predicted.

label_column : unicode

Column containing the actual label for each observation.

observation_columns : list (default=None)

Column(s) containing the observations whose labels are to be predicted and tested.
Default is to test over the columns the `SvmModel` was trained on.

Returns : dict

object

An object with binary classification metrics. The data returned is composed of multiple components:

```
<object>.accuracy : double <object>.confusion_matrix : table <object>.f_measure :
double <object>.precision : double <object>.recall : double
```

Predict the labels for a test frame and run classification metrics on predicted and target labels.

27.6.5 *SvmModel* train

```
train(self, frame, label_column, observation_columns, intercept=None, num_iterations=None,
step_size=None, reg_type=None, reg_param=None, mini_batch_fraction=None)
[ALPHA] Train SVM model based on another frame.
```

Parameters **frame** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame to train the model on.

label_column : unicode

Column name containing the label for each observation.

observation_columns : list

Column(s) containing the observations.

intercept : bool (default=None)

The algorithm adds an intercept. Default is true.

num_iterations : int32 (default=None)

Number of iterations. Default is 100.

step_size : int32 (default=None)

Step size for optimizer. Default is 1.0.

reg_type : unicode (default=None)

Regularization L1 or L2. Default is L2.

reg_param : float64 (default=None)

Regularization parameter. Default is 0.01.

mini_batch_fraction : float64 (default=None)

Mini batch fraction parameter. Default is 1.0.

Returns : _Unit

Creating a SVM Model using the observation column and label column of the train frame.

```
class SvmModel
create a new model
```

Attributes

name	Set or get the name of the model object.
-------------	--

Methods

<code>__init__(self[, name, _info])</code>	[ALPHA] create a new model
<code>predict(self, frame[, observation_columns])</code>	[ALPHA] Make new frame with additional column for predicted label.
<code>test(self, frame, label_column[, observation_columns])</code>	[ALPHA] Predict test frame labels and return metrics.
<code>train(self, frame, label_column, observation_columns[, intercept, ...])</code>	[ALPHA] Train SVM model based on another frame.

`__init__ (self, name=None)`
[ALPHA] create a new model

Parameters **name** : unicode (default=None)

User supplied name.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

27.7 Models LdaModel

27.7.1 LdaModel new

`__init__ (self, name=None)`
Creates Latent Dirichlet Allocation model

Parameters **name** : unicode (default=None)

User supplied name.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

Topic Modeling with Latent Dirichlet Allocation

Topic modeling algorithms are a class of statistical approaches to partitioning items in a data set into subgroups. As the name implies, these algorithms are often used on corpora of textual data, where they are used to group documents in the collection into semantically-meaningful groupings. For an overall introduction to topic modeling, the reader might refer to the work of David Blei and Michael Jordan, who are credited with creating and popularizing topic modeling in the machine learning community. In particular, Blei’s 2011 paper provides a nice introduction, and is freely-available online ⁷.

LDA (Latent Dirichlet Allocation) is a commonly-used algorithm for topic modeling, but, more broadly, is considered a dimensionality reduction technique. It contrasts with other approaches (for example, latent semantic indexing), in that it creates what’s referred to as a generative probabilistic model — a statistical model that allows the algorithm to generalize its approach to topic assignment to other, never-before-seen data points. For the purposes of exposition, we’ll limit the scope of our discussion of LDA to the world of natural language processing, as it has an intuitive use there (though LDA can be used on other types of data). In general, LDA represents documents as random mixtures over topics in the corpus. This makes sense because any work of writing is rarely about a single subject. Take the case of a news article on the President of the

⁷ <http://www.cs.princeton.edu/~blei/papers/Blei2011.pdf>

United States of America’s approach to healthcare as an example. It would be reasonable to assign topics like President, USA, health insurance, politics, or healthcare to such a work, though it is likely to primarily discuss the President and healthcare.

LDA assumes that input corpora contain documents pertaining to a given number of topics, each of which are associated with a variety of words, and that each document is the result of a mixture of probabilistic samplings: first over the distribution of possible topics for the corpora, and second over the list of possible words in the selected topic. This generative assumption confers one of the main advantages LDA holds over other topic modeling approaches, such as probabilistic and regular LSI (Latent Semantic Indexing). As a generative model, LDA is able to generalize the model it uses to separate documents into topics to documents outside the corpora. For example, this means that using LDA to group online news articles into categories like Sports, Entertainment, and Politics, it would be possible to use the fitted model to help categorize newly-published news stories. Such an application is beyond the scope of approaches like LSI. What’s more, when fitting an LSI model, the number of parameters that have to be estimated scale linearly with the number of documents in the corpus, whereas the number of parameters to estimate for an LDA model scales with the number of topics — a much lower number, making it much better-suited to working with large data sets.

The Typical Latent Dirichlet Allocation Workflow

Although every user is likely to have his or her own habits and preferred approach to topic modeling a document corpus, there is a general workflow that is a good starting point when working with new data. The general steps to the topic modeling with LDA include:

- 1.Data preparation and ingest
- 2.Assignment to training or testing partition
- 3.Graph construction
- 4.Training LDA
- 5.Evaluation
- 6.Interpretation of results

Data preparation and ingest

Most topic modeling workflows involve several data pre-processing and cleaning steps. Depending on the characteristics of the data being analyzed, there are different best-practices to use here, so it’s important to be familiar with the standard procedures for analytics in the domain from which the text originated. For example, in the biomedical text analytics community, it is common practice for text analytics workflows to involve pre-processing for identifying negation statements (Chapman et al., 2001⁸). The reason for this is many analysts in that domain are examining text for diagnostic statements — thus, failing to identify a negated statement in which a disease is mentioned could lead to undesirable false-positives, but this phenomenon may not arise in every domain. In general, both stemming and stop word filtering are recommended steps for topic modeling pre-processing. Stemming refers to a set of methods used to normalize different tenses and variations of the same word (for example, stemmer, stemming, stemmed, and stem). Stemming algorithms will normalize all variations of a word to one common form (for example, stem). There are many approaches to stemming, but the Porter Stemming (Porter, 2006⁹) is one of the most commonly-used.

Removing common, uninformative words, or stop word filtering, is another commonly-used step in data pre-processing for topic modeling. Stop words include words like *the*, *and*, or *a*, but the full list of uninformative words can be quite long and depend on the domain producing the text in question. Example stop word lists online¹⁰ can be a great place to start, but being aware of the best-practices in the applicable field is necessary to expand upon these.

⁸ <http://www.sciencedirect.com/science/article/pii/S1532046401910299>

⁹ <http://tartarus.org/~martin/PorterStemmer/index.html>

¹⁰ <http://www.textfixer.com/resources/common-english-words.txt>

There may be other pre-processing steps needed, depending on the type of text being worked with. Punctuation removal is frequently recommended, for example. To determine what's best for the text being analyzed, it helps to understand a bit about how LDA analyzes the input text. To learn the topic model, LDA will typically look at the frequency of individual words across documents, which are determined based on space-separation. Thus, each word will be interpreted independent of where it occurs in a document, and without regard for the words that were written around it. In the text analytics field, this is often referred to as a *bag of words* approach to tokenization, the process of separating input text into composite features to be analyzed by some algorithm. When choosing pre-processing steps, it helps to keep this in mind. Don't worry too much about removing words or modifying their format — you're not manipulating your data! These steps simply make it easier for the topic modeling algorithm to find the latent topics that comprise your corpus.

Assignment to training or testing partition

The random assignment to training and testing partitions is an important step in most every machine learning workflow. It is common practice to withhold a random selection of one's data set for the purpose of evaluating the accuracy of the model that was learned from the training data. The results of this evaluation allow the user to confidently speak about the generalizability of the trained model. When speaking in these terms, be cautious that you only discuss generalizability to the broader population from which your data was originally obtained. If a topic model is trained on neuroscience-related publications, for example, evaluating the model on other neuroscience-related publications is valid. It would not be valid to discuss the model's ability to work on documents from other domains.

There are various schools of thought for how to assign a data set to training and testing collections, but all agree that the process should be random. Where analysts disagree is in the ratio of data to be assigned to each. In most situations, the bulk of data will be assigned to the training collection, because the more data that can be used to train the algorithm, the better the resultant model will typically be. It's also important that the testing collection have sufficient data to be able to reflect the characteristics of the larger population from which it was drawn (this becomes an important issue when working with data sets with rare topics, for example). As a starting point, many people will use a 90%/10% training/test collection split, and modify this ratio based on the characteristics of the documents being analyzed.

Graph construction

Trusted Analytics uses a bipartite graph, to learn an LDA topic model. This graph contains vertices in two columns. The left-hand column contains unique ids, each corresponding to a document in the training collection, while the right-hand column contains unique ids corresponding to each word in the entire training set, following any pre-processing steps that were used. Connections between these columns, or edges, denote the number of times a particular word appears in a document, with the weight on the edge in question denoting the number of times the word was found there. After graph construction, many analysts choose to normalize the weights using one of a variety of normalization schemes. One approach is to normalize the weights to sum to 1, while another is to use an approach called term frequency-inverse document frequency (tfidf), where the resultant weights are meant to reflect how important a word is to a document in the corpus. Whether to use normalization — or what technique to use — is an open question, and will likely depend on the characteristics of the text being analyzed. Typical text analytics experiments will try a variety of approaches on a small subset of the data to determine what works best.

See [Figure 1](#).

Training the Model

In using LDA, we are trying to model a document collection in terms of topics $\beta_{1:K}$, where each β_K describes a distribution over the set of words in the training corpus. Every document d , then, is a vector of proportions θ_d , where $\theta_{d,k}$ is the proportion of the d^{th} document for topic k . The topic assignment for document d is z_d , and $z_{d,n}$ is the topic assignment for the n^{th} word in document d . The words observed in document d are $w_{-}[d]$, and $w_{d,n}$ is the n^{th} word in document d . The generative process for LDA, then, is the joint

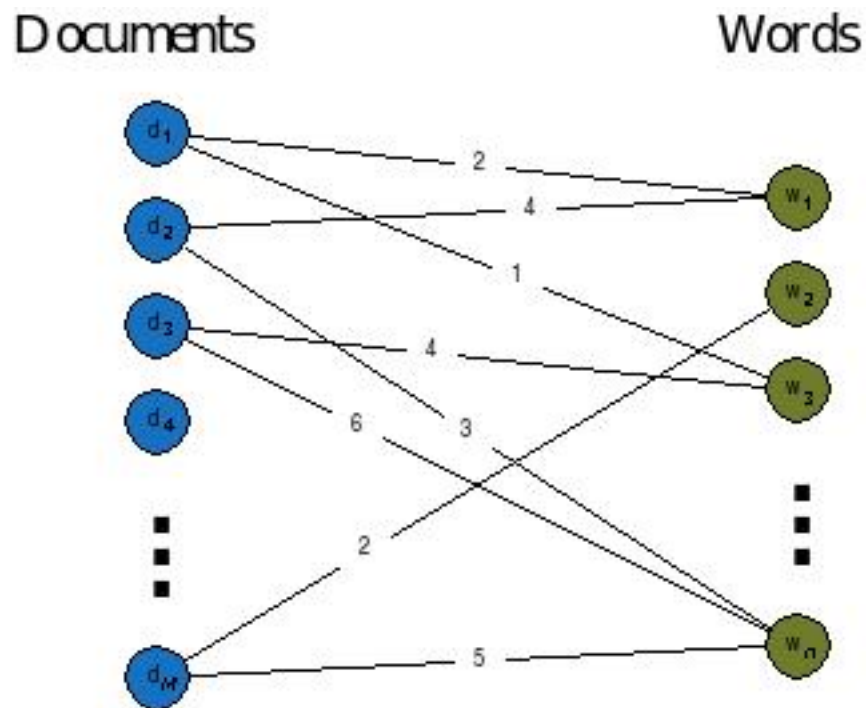


Fig. 27.4: Figure 1 - Example layout of a bipartite graph for LDA.

The left-hand column contains one vertex for each document in the input corpus, while the right-hand column contains vertices for each unique word found in them. Edges connecting left- and right-hand columns denote the number of times the word was found in the document the edge connects. The weights of the edges used in this example were not normalized.

distribution of hidden and observed values

$$p(\beta_{1:K}, \theta_{1:D}, z_{1:D}, w_{1:D}) = \prod_{i=1}^K p(\beta_i) \prod_{d=1}^D p(\theta_d) \left(\prod_{n=1}^N p(z_{d,n} | \theta_d) p(w_{d,n} | \beta_{1:K}, z_{d,n}) \right)$$

This distribution depicts several dependencies: topic assignment $z_{d,n}$ depends on the topic proportions θ_d , and the observed word $w_{d,n}$ depends on topic assignment $z_{d,n}$ and all the topics $\beta_{1:K}$, for example. Although there are no analytical solutions to learning the LDA model, there are a variety of approximate solutions that are used, most of which are based on Gibbs Sampling (for example, Porteous et al., 2008¹¹). The Trusted Analytics uses an implementation related to this. We refer the interested reader to the primary source on this approach to learn more (Teh et al., 2006¹²).

Evaluation

As with every machine learning algorithm, evaluating the accuracy of the model that has been obtained is an important step before interpreting the results. With many types of algorithms, the best practices in this step are straightforward — in supervised classification, for example, we know the true labels of the data being classified, so evaluating performance can be as simple as computing the number of errors, calculating receiver operating characteristic, or F1 measure. With topic modeling, the situation is not so straightforward. This makes sense, if we consider with LDA we’re using an algorithm to blindly identify logical subgroupings in our data, and we don’t *a priori* know the best grouping that can be found. Evaluation, then, should proceed with this in mind, and an examination of homogeneity of the words comprising the documents in each grouping is often done. This issue is discussed further in Blei’s 2011 introduction to topic modeling¹³. It is of course possible to evaluate a topic model from a statistical perspective using our hold-out testing document collection — and this is a recommended best practice — however, such an evaluation does not assess the topic model in terms of how they are typically used.

Interpretation of results

After running LDA on a document corpus, users will typically examine the top n most frequent words that can be found in each grouping. With this information, one is often able to use their own domain expertise to think of logical names for each topic (this situation is analogous to the step in principal components analysis, wherein statisticians will think of logical names for each principal component based on the mixture of dimensions each spans). Each document, then, can be assigned to a topic, based on the mixture of topics it has been assigned. Recall that LDA will assign each document a set of probabilities corresponding to each possible topic. Researchers will often set some threshold value to make a categorical judgment regarding topic membership, using this information.

footnotes

27.7.2 *LdaModel* name

name

Set or get the name of the model object.

Parameters

Change or retrieve model object identification. Identification names must start with a letter and are limited to alphanumeric characters and the `_` character.

¹¹ <http://www.ics.uci.edu/~newman/pubs/fastlda.pdf>

¹² http://machinelearning.wustl.edu/mlpapers/paper_files/NIPS2006_511.pdf

¹³ <http://www.cs.princeton.edu/~blei/papers/Blei2011.pdf>

Examples

```
>>> my_model.name

"csv_data"

>>> my_model.name = "cleaned_data"
>>> my_model.name

"cleaned_data"
```

27.7.3 *LdaModel* predict

predict (*self*, *document*)

[BETA] Predict conditional probabilities of topics given document.

Parameters **document** : list

Document whose topics are to be predicted.

Returns : dict

dict Dictionary containing predicted topics. The data returned is composed of multiple components:

topics_given_doc [list of doubles] List of conditional probabilities of topics given document.

new_words_count [int] Count of new words in test document not present in training set.

new_words_percentage: double Percentage of new words in test document.

Predicts conditional probabilities of topics given document using trained Latent Dirichlet Allocation model. The input document is represented as a list of strings

27.7.4 *LdaModel* publish

publish (*self*)

[BETA] Creates a tar file that will be used as input to the scoring engine

Parameters

Returns : dict

Returns the HDFS path to the tar file

Creates a tar file with the trained Latent Dirichlet Allocation model. The tar file is used as input to the scoring engine to predict the conditional topic probabilities for a document.

27.7.5 *LdaModel* train

```
train (self, frame, document_column_name, word_column_name, word_count_column_name,  
       max_iterations=None, alpha=None, beta=None, convergence_threshold=None, evaluate_cost=None,  
       num_topics=None)
```

[BETA] Creates Latent Dirichlet Allocation model

Parameters frame : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

Input frame data.

document_column_name : unicode

Column Name for documents. Column should contain a str value.

word_column_name : unicode

Column name for words. Column should contain a str value.

word_count_column_name : unicode

Column name for word count. Column should contain an int32 or int64 value.

max_iterations : int32 (default=None)

The maximum number of iterations that the algorithm will execute. The valid value range is all positive int. Default is 20.

alpha : float32 (default=None)

The hyper-parameter for document-specific distribution over topics. Mainly used as a smoothing parameter in *Bayesian inference*. Larger value implies that documents are assumed to cover all topics more uniformly; smaller value implies that documents are more concentrated on a small subset of topics. Valid value range is all positive float.

Default is 0.1.

beta : float32 (default=None)

The hyper-parameter for word-specific distribution over topics. Mainly used as a smoothing parameter in *Bayesian inference*. Larger value implies that topics contain all words more uniformly and smaller value implies that topics are more concentrated on a small subset of words. Valid value range is all positive float. Default is 0.1.

convergence_threshold : float32 (default=None)

The amount of change in LDA model parameters that will be tolerated at convergence. If the change is less than this threshold, the algorithm exits before it reaches the maximum number of supersteps. Valid value range is all positive float and 0.0. Default is 0.001.

evaluate_cost : bool (default=None)

“True” means turn on cost evaluation and “False” means turn off cost evaluation. It’s relatively expensive for LDA to evaluate cost function. For time-critical applications, this option allows user to turn off cost function evaluation. Default is “False”.

num_topics : int32 (default=None)

The number of topics to identify in the LDA model. Using fewer topics will speed up the computation, but the extracted topics might be more abstract or less specific; using

more topics will result in more computation but lead to more specific topics. Valid value range is all positive int. Default is 10.

Returns : dict

dict The data returned is composed of multiple components:

topics_given_doc [Frame] Frame with conditional probabilities of topic given document.

word_given_topics [Frame] Frame with conditional probabilities of word given topic.

topics_given_word [Frame] Frame with conditional probabilities of topic given word.

report [str] The configuration and learning curve report for Latent Dirichlet Allocation as a multiple line str.

See the discussion about [Latent Dirichlet Allocation at Wikipedia](http://en.wikipedia.org/wiki/Latent_Dirichlet_allocation).¹⁴

class LdaModel

Creates Latent Dirichlet Allocation model

Topic Modeling with Latent Dirichlet Allocation

Topic modeling algorithms are a class of statistical approaches to partitioning items in a data set into subgroups. As the name implies, these algorithms are often used on corpora of textual data, where they are used to group documents in the collection into semantically-meaningful groupings. For an overall introduction to topic modeling, the reader might refer to the work of David Blei and Michael Jordan, who are credited with creating and popularizing topic modeling in the machine learning community. In particular, Blei’s 2011 paper provides a nice introduction, and is freely-available online [\[#LDA1\]](#).

LDA is a commonly-used algorithm for topic modeling, but, more broadly, is considered a dimensionality reduction technique. It contrasts with other approaches (for example, latent semantic indexing), in that it creates what’s referred to as a generative probabilistic model — a statistical model that allows the algorithm to generalize its approach to topic assignment to other, never-before-seen data points. For the purposes of exposition, we’ll limit the scope of our discussion of LDA to the world of natural language processing, as it has an intuitive use there (though LDA can be used on other types of data). In general, LDA represents documents as random mixtures over topics in the corpus. This makes sense because any work of writing is rarely about a single subject. Take the case of a news article on the President of the United States of America’s approach to healthcare as an example. It would be reasonable to assign topics like President, USA, health insurance, politics, or healthcare to such a work, though it is likely to primarily discuss the President and healthcare.

LDA assumes that input corpora contain documents pertaining to a given number of topics, each of which are associated with a variety of words, and that each document is the result of a mixture of probabilistic samplings: first over the distribution of possible topics for the corpora, and second over the list of possible words in the selected topic. This generative assumption confers one of the main advantages LDA holds over other topic modeling approaches, such as probabilistic and regular LSI. As a generative model, LDA is able to generalize the model it uses to separate documents into topics to documents outside the corpora. For example, this means that using LDA to group online news articles into categories like Sports, Entertainment, and Politics, it would be possible to use the fitted model to help categorize newly-published news stories. Such an application is beyond the scope of approaches like LSI. What’s more, when fitting an LSI model, the number of parameters that have to be estimated scale linearly with the number of documents in the corpus, whereas the number of parameters to estimate for an LDA model scales with the number of topics — a much lower number, making it much better-suited to working with large data sets.

The Typical Latent Dirichlet Allocation Workflow

Although every user is likely to have his or her own habits and preferred approach to topic modeling a document corpus, there is a general workflow that is a good starting point when working with new data. The

¹⁴http://en.wikipedia.org/wiki/Latent_Dirichlet_allocation

general steps to the topic modeling with LDA include:

- 1.Data preparation and ingest
- 2.Assignment to training or testing partition
- 3.Graph construction
- 4.Training LDA
- 5.Evaluation
- 6.Interpretation of results

Data preparation and ingest

Most topic modeling workflows involve several data pre-processing and cleaning steps. Depending on the characteristics of the data being analyzed, there are different best-practices to use here, so it's important to be familiar with the standard procedures for analytics in the domain from which the text originated. For example, in the biomedical text analytics community, it is common practice for text analytics workflows to involve pre-processing for identifying negation statements (Chapman et al., 2001 [\[#LDA2\]](#)). The reason for this is many analysts in that domain are examining text for diagnostic statements — thus, failing to identify a negated statement in which a disease is mentioned could lead to undesirable false-positives, but this phenomenon may not arise in every domain. In general, both stemming and stop word filtering are recommended steps for topic modeling pre-processing. Stemming refers to a set of methods used to normalize different tenses and variations of the same word (for example, stemmer, stemming, stemmed, and stem). Stemming algorithms will normalize all variations of a word to one common form (for example, stem). There are many approaches to stemming, but the Porter Stemming (Porter, 2006 [\[#LDA3\]](#)) is one of the most commonly-used.

Removing common, uninformative words, or stop word filtering, is another commonly-used step in data pre-processing for topic modeling. Stop words include words like *the*, *and*, or *a*, but the full list of uninformative words can be quite long and depend on the domain producing the text in question. Example stop word lists online [\[#LDA4\]](#) can be a great place to start, but being aware of the best-practices in the applicable field is necessary to expand upon these.

There may be other pre-processing steps needed, depending on the type of text being worked with. Punctuation removal is frequently recommended, for example. To determine what's best for the text being analyzed, it helps to understand a bit about how LDA analyzes the input text. To learn the topic model, LDA will typically look at the frequency of individual words across documents, which are determined based on space-separation. Thus, each word will be interpreted independent of where it occurs in a document, and without regard for the words that were written around it. In the text analytics field, this is often referred to as a *bag of words* approach to tokenization, the process of separating input text into composite features to be analyzed by some algorithm. When choosing pre-processing steps, it helps to keep this in mind. Don't worry too much about removing words or modifying their format — you're not manipulating your data! These steps simply make it easier for the topic modeling algorithm to find the latent topics that comprise your corpus.

Assignment to training or testing partition

The random assignment to training and testing partitions is an important step in most every machine learning workflow. It is common practice to withhold a random selection of one's data set for the purpose of evaluating the accuracy of the model that was learned from the training data. The results of this evaluation allow the user to confidently speak about the generalizability of the trained model. When speaking in these terms, be cautious that you only discuss generalizability to the broader population from which your data was originally obtained. If a topic model is trained on neuroscience-related publications, for example, evaluating the model on other neuroscience-related publications is valid. It would not be valid to discuss the model's ability to work on documents from other domains.

There are various schools of thought for how to assign a data set to training and testing collections, but all agree that the process should be random. Where analysts disagree is in the ratio of data to be assigned to each. In most situations, the bulk of data will be assigned to the training collection, because the more data that can be

used to train the algorithm, the better the resultant model will typically be. It's also important that the testing collection have sufficient data to be able to reflect the characteristics of the larger population from which it was drawn (this becomes an important issue when working with data sets with rare topics, for example). As a starting point, many people will use a 90%/10% training/test collection split, and modify this ratio based on the characteristics of the documents being analyzed.

Graph construction

Trusted Analytics uses a bipartite graph, to learn an LDA topic model. This graph contains vertices in two columns. The left-hand column contains unique ids, each corresponding to a document in the training collection, while the right-hand column contains unique ids corresponding to each word in the entire training set, following any pre-processing steps that were used. Connections between these columns, or edges, denote the number of times a particular word appears in a document, with the weight on the edge in question denoting the number of times the word was found there. After graph construction, many analysts choose to normalize the weights using one of a variety of normalization schemes. One approach is to normalize the weights to sum to 1, while another is to use an approach called term frequency-inverse document frequency (tfidf), where the resultant weights are meant to reflect how important a word is to a document in the corpus. Whether to use normalization — or what technique to use — is an open question, and will likely depend on the characteristics of the text being analyzed. Typical text analytics experiments will try a variety of approaches on a small subset of the data to determine what works best.

See [Figure 1](#).

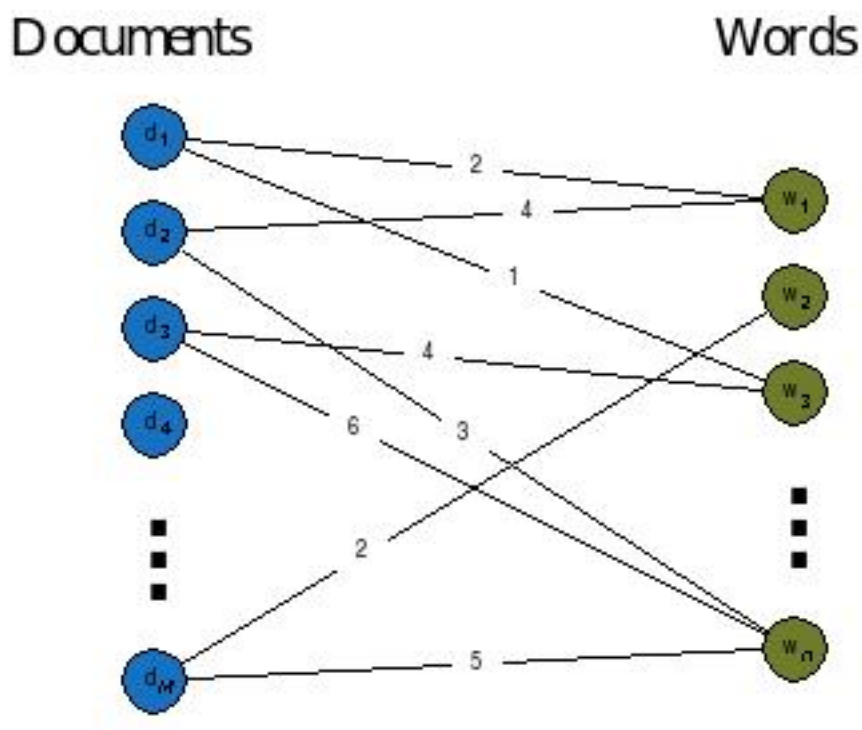


Fig. 27.5: Figure 1 - Example layout of a bipartite graph for LDA.

The left-hand column contains one vertex for each document in the input corpus, while the right-hand column contains vertices for each unique word found in them. Edges connecting left- and right-hand columns denote the number of times the word was found in the document the edge connects. The weights of the edges used in this example were not normalized.

Training the Model

In using LDA, we are trying to model a document collection in terms of topics $\beta_{1:K}$, where each β_K describes

a distribution over the set of words in the training corpus. Every document d , then, is a vector of proportions θ_d , where $\theta_{d,k}$ is the proportion of the d^{th} document for topic k . The topic assignment for document d is z_d , and $z_{d,n}$ is the topic assignment for the n^{th} word in document d . The words observed in document d are $w_{d,n}$, and $w_{d,n}$ is the n^{th} word in document d . The generative process for LDA, then, is the joint distribution of hidden and observed values

$$p(\beta_{1:K}, \theta_{1:D}, z_{1:D}, w_{1:D}) = \prod_{i=1}^K p(\beta_i) \prod_{i=1}^D p(\theta_d) \left(\prod_{n=1}^N p(z_{d,n} | \theta_d) p(w_{d,n} | \beta_{1:K}, z_{d,n}) \right)$$

This distribution depicts several dependencies: topic assignment $z_{d,n}$ depends on the topic proportions θ_d , and the observed word $w_{d,n}$ depends on topic assignment $z_{d,n}$ and all the topics $\beta_{1:K}$, for example. Although there are no analytical solutions to learning the LDA model, there are a variety of approximate solutions that are used, most of which are based on Gibbs Sampling (for example, Porteous et al., 2008 [\[LDA5\]](#)). The Trusted Analytics uses an implementation related to this. We refer the interested reader to the primary source on this approach to learn more (Teh et al., 2006 [\[LDA6\]](#)).

Evaluation

As with every machine learning algorithm, evaluating the accuracy of the model that has been obtained is an important step before interpreting the results. With many types of algorithms, the best practices in this step are straightforward — in supervised classification, for example, we know the true labels of the data being classified, so evaluating performance can be as simple as computing the number of errors, calculating receiver operating characteristic, or F1 measure. With topic modeling, the situation is not so straightforward. This makes sense, if we consider with LDA we’re using an algorithm to blindly identify logical subgroupings in our data, and we don’t *a priori* know the best grouping that can be found. Evaluation, then, should proceed with this in mind, and an examination of homogeneity of the words comprising the documents in each grouping is often done. This issue is discussed further in Blei’s 2011 introduction to topic modeling [\[LDA7\]](#). It is of course possible to evaluate a topic model from a statistical perspective using our hold-out testing document collection — and this is a recommended best practice — however, such an evaluation does not assess the topic model in terms of how they are typically used.

Interpretation of results

After running LDA on a document corpus, users will typically examine the top n most frequent words that can be found in each grouping. With this information, one is often able to use their own domain expertise to think of logical names for each topic (this situation is analogous to the step in principal components analysis, wherein statisticians will think of logical names for each principal component based on the mixture of dimensions each spans). Each document, then, can be assigned to a topic, based on the mixture of topics it has been assigned. Recall that LDA will assign each document a set of probabilities corresponding to each possible topic. Researchers will often set some threshold value to make a categorical judgment regarding topic membership, using this information.

footnotes

Attributes

name	Set or get the name of the model object.
------	--

Methods

<code>__init__(self[, name, _info])</code>	Creates Latent Dirichlet Allocation model
<code>predict(self, document)</code>	[BETA] Predict conditional probabilities of topics given document.
<code>publish(self)</code>	[BETA] Creates a tar file that will used as input to the scoring engine
<code>train(self, frame, document_column_name, word_column_name, word_count_column_name)</code>	[BETA] Creates Latent Dirichlet Allocation model

`__init__(self, name=None)`

Creates Latent Dirichlet Allocation model

Parameters `name` : unicode (default=None)

User supplied name.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

Topic Modeling with Latent Dirichlet Allocation

Topic modeling algorithms are a class of statistical approaches to partitioning items in a data set into subgroups. As the name implies, these algorithms are often used on corpora of textual data, where they are used to group documents in the collection into semantically-meaningful groupings. For an overall introduction to topic modeling, the reader might refer to the work of David Blei and Michael Jordan, who are credited with creating and popularizing topic modeling in the machine learning community. In particular, Blei’s 2011 paper provides a nice introduction, and is freely-available online [\[#LDA1\]](#).

LDA is a commonly-used algorithm for topic modeling, but, more broadly, is considered a dimensionality reduction technique. It contrasts with other approaches (for example, latent semantic indexing), in that it creates what’s referred to as a generative probabilistic model — a statistical model that allows the algorithm to generalize its approach to topic assignment to other, never-before-seen data points. For the purposes of exposition, we’ll limit the scope of our discussion of LDA to the world of natural language processing, as it has an intuitive use there (though LDA can be used on other types of data). In general, LDA represents documents as random mixtures over topics in the corpus. This makes sense because any work of writing is rarely about a single subject. Take the case of a news article on the President of the United States of America’s approach to healthcare as an example. It would be reasonable to assign topics like President, USA, health insurance, politics, or healthcare to such a work, though it is likely to primarily discuss the President and healthcare.

LDA assumes that input corpora contain documents pertaining to a given number of topics, each of which are associated with a variety of words, and that each document is the result of a mixture of probabilistic samplings: first over the distribution of possible topics for the corpora, and second over the list of possible words in the selected topic. This generative assumption confers one of the main advantages LDA holds over other topic modeling approaches, such as probabilistic and regular LSI. As a generative model, LDA is able to generalize the model it uses to separate documents into topics to documents outside the corpora. For example, this means that using LDA to group online news articles into categories like Sports, Entertainment, and Politics, it would be possible to use the fitted model to help categorize newly-published news stories. Such an application is beyond the scope of approaches like LSI. What’s more, when fitting an LSI model, the number of parameters that have to be estimated scale linearly with the number of documents in the corpus, whereas the number of parameters to estimate for an LDA model scales with the number of topics — a much lower number, making it much better-suited to working with large data sets.

The Typical Latent Dirichlet Allocation Workflow

Although every user is likely to have his or her own habits and preferred approach to topic modeling a document corpus, there is a general workflow that is a good starting point when working with new data. The general steps to the topic modeling with LDA include:

- 1.Data preparation and ingest
- 2.Assignment to training or testing partition
- 3.Graph construction
- 4.Training LDA
- 5.Evaluation
- 6.Interpretation of results

Data preparation and ingest

Most topic modeling workflows involve several data pre-processing and cleaning steps. Depending on the characteristics of the data being analyzed, there are different best-practices to use here, so it's important to be familiar with the standard procedures for analytics in the domain from which the text originated. For example, in the biomedical text analytics community, it is common practice for text analytics workflows to involve pre-processing for identifying negation statements (Chapman et al., 2001 [\[#LDA2\]](#)). The reason for this is many analysts in that domain are examining text for diagnostic statements — thus, failing to identify a negated statement in which a disease is mentioned could lead to undesirable false-positives, but this phenomenon may not arise in every domain. In general, both stemming and stop word filtering are recommended steps for topic modeling pre-processing. Stemming refers to a set of methods used to normalize different tenses and variations of the same word (for example, stemmer, stemming, stemmed, and stem). Stemming algorithms will normalize all variations of a word to one common form (for example, stem). There are many approaches to stemming, but the Porter Stemming (Porter, 2006 [\[#LDA3\]](#)) is one of the most commonly-used.

Removing common, uninformative words, or stop word filtering, is another commonly-used step in data pre-processing for topic modeling. Stop words include words like *the*, *and*, or *a*, but the full list of uninformative words can be quite long and depend on the domain producing the text in question. Example stop word lists online [\[#LDA4\]](#) can be a great place to start, but being aware of the best-practices in the applicable field is necessary to expand upon these.

There may be other pre-processing steps needed, depending on the type of text being worked with. Punctuation removal is frequently recommended, for example. To determine what's best for the text being analyzed, it helps to understand a bit about how LDA analyzes the input text. To learn the topic model, LDA will typically look at the frequency of individual words across documents, which are determined based on space-separation. Thus, each word will be interpreted independent of where it occurs in a document, and without regard for the words that were written around it. In the text analytics field, this is often referred to as a *bag of words* approach to tokenization, the process of separating input text into composite features to be analyzed by some algorithm. When choosing pre-processing steps, it helps to keep this in mind. Don't worry too much about removing words or modifying their format — you're not manipulating your data! These steps simply make it easier for the topic modeling algorithm to find the latent topics that comprise your corpus.

Assignment to training or testing partition

The random assignment to training and testing partitions is an important step in most every machine learning workflow. It is common practice to withhold a random selection of one's data set for the purpose of evaluating the accuracy of the model that was learned from the training data. The results of this evaluation allow the user to confidently speak about the generalizability of the trained model. When speaking in these terms, be cautious that you only discuss generalizability to the broader population from which your data was originally obtained. If a topic model is trained on neuroscience-related publications, for example, evaluating the model on other neuroscience-related publications is valid. It would not be valid to discuss the model's ability to work on documents from other domains.

There are various schools of thought for how to assign a data set to training and testing collections, but all agree that the process should be random. Where analysts disagree is in the ratio of data to be assigned to each. In most situations, the bulk of data will be assigned to the training collection, because the more data that can be used to train the algorithm, the better the resultant model will typically be. It's also important that the testing collection have sufficient data to be able to reflect the characteristics of the larger population from which it was

drawn (this becomes an important issue when working with data sets with rare topics, for example). As a starting point, many people will use a 90%/10% training/test collection split, and modify this ratio based on the characteristics of the documents being analyzed.

Graph construction

Trusted Analytics uses a bipartite graph, to learn an LDA topic model. This graph contains vertices in two columns. The left-hand column contains unique ids, each corresponding to a document in the training collection, while the right-hand column contains unique ids corresponding to each word in the entire training set, following any pre-processing steps that were used. Connections between these columns, or edges, denote the number of times a particular word appears in a document, with the weight on the edge in question denoting the number of times the word was found there. After graph construction, many analysts choose to normalize the weights using one of a variety of normalization schemes. One approach is to normalize the weights to sum to 1, while another is to use an approach called term frequency-inverse document frequency (tfidf), where the resultant weights are meant to reflect how important a word is to a document in the corpus. Whether to use normalization — or what technique to use — is an open question, and will likely depend on the characteristics of the text being analyzed. Typical text analytics experiments will try a variety of approaches on a small subset of the data to determine what works best.

See [Figure 1](#).

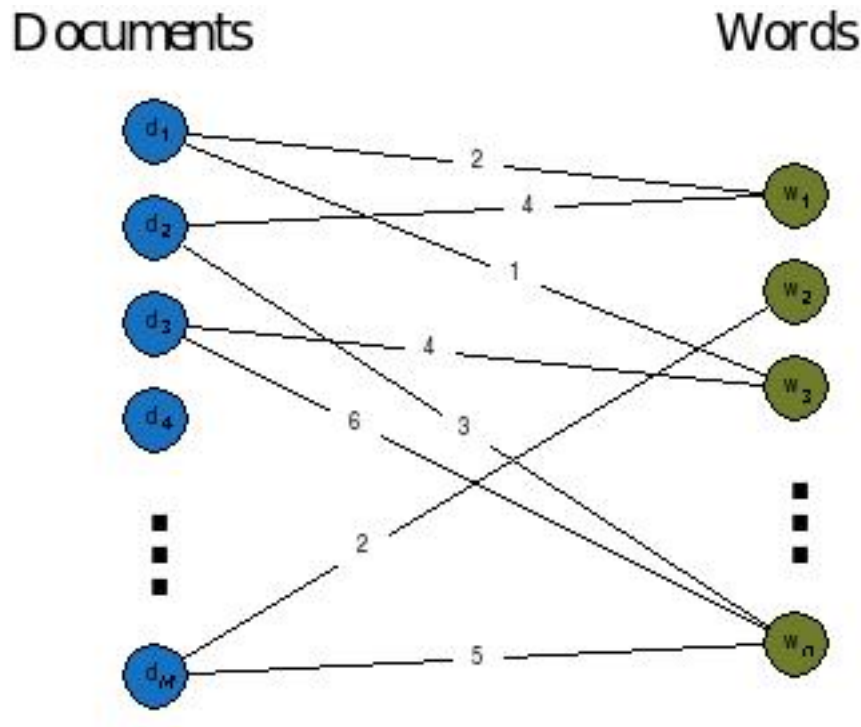


Fig. 27.6: Figure 1 - Example layout of a bipartite graph for LDA.

The left-hand column contains one vertex for each document in the input corpus, while the right-hand column contains vertices for each unique word found in them. Edges connecting left- and right-hand columns denote the number of times the word was found in the document the edge connects. The weights of the edges used in this example were not normalized.

Training the Model

In using LDA, we are trying to model a document collection in terms of topics $\beta_{1:K}$, where each β_K describes a distribution over the set of words in the training corpus. Every document d , then, is a vector of proportions θ_d , where $\theta_{d,k}$ is the proportion of the d^{th} document for topic k . The topic assignment for document d is z_d ,

and $z_{d,n}$ is the topic assignment for the n^{th} word in document d . The words observed in document d are $w_{d,1}, \dots, w_{d,N}$, and $w_{d,n}$ is the n^{th} word in document d . The generative process for LDA, then, is the joint distribution of hidden and observed values

$$p(\beta_{1:K}, \theta_{1:D}, z_{1:D}, w_{1:D}) = \prod_{i=1}^K p(\beta_i) \prod_{d=1}^D p(\theta_d) \left(\prod_{n=1}^N p(z_{d,n} | \theta_d) p(w_{d,n} | \beta_{1:K}, z_{d,n}) \right)$$

This distribution depicts several dependencies: topic assignment $z_{d,n}$ depends on the topic proportions θ_d , and the observed word $w_{d,n}$ depends on topic assignment $z_{d,n}$ and all the topics $\beta_{1:K}$, for example. Although there are no analytical solutions to learning the LDA model, there are a variety of approximate solutions that are used, most of which are based on Gibbs Sampling (for example, Porteous et al., 2008 [\[#LDA5\]](#)). The Trusted Analytics uses an implementation related to this. We refer the interested reader to the primary source on this approach to learn more (Teh et al., 2006 [\[#LDA6\]](#)).

Evaluation

As with every machine learning algorithm, evaluating the accuracy of the model that has been obtained is an important step before interpreting the results. With many types of algorithms, the best practices in this step are straightforward — in supervised classification, for example, we know the true labels of the data being classified, so evaluating performance can be as simple as computing the number of errors, calculating receiver operating characteristic, or F1 measure. With topic modeling, the situation is not so straightforward. This makes sense, if we consider with LDA we’re using an algorithm to blindly identify logical subgroupings in our data, and we don’t *a priori* know the best grouping that can be found. Evaluation, then, should proceed with this in mind, and an examination of homogeneity of the words comprising the documents in each grouping is often done. This issue is discussed further in Blei’s 2011 introduction to topic modeling [\[#LDA7\]](#). It is of course possible to evaluate a topic model from a statistical perspective using our hold-out testing document collection — and this is a recommended best practice — however, such an evaluation does not assess the topic model in terms of how they are typically used.

Interpretation of results

After running LDA on a document corpus, users will typically examine the top n most frequent words that can be found in each grouping. With this information, one is often able to use their own domain expertise to think of logical names for each topic (this situation is analogous to the step in principal components analysis, wherein statisticians will think of logical names for each principal component based on the mixture of dimensions each spans). Each document, then, can be assigned to a topic, based on the mixture of topics it has been assigned. Recall that LDA will assign each document a set of probabilities corresponding to each possible topic. Researchers will often set some threshold value to make a categorical judgment regarding topic membership, using this information.

footnotes

27.8 Models LogisticRegressionModel

27.8.1 LogisticRegressionModel new

`__init__` (*self*, *name=None*)
Create a ‘new’ instance of logistic regression model.

Parameters *name* : unicode (default=None)

User supplied name.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

27.8.2 *LogisticRegressionModel* name

name

Set or get the name of the model object.

Parameters

Change or retrieve model object identification. Identification names must start with a letter and are limited to alphanumeric characters and the `_` character.

Examples

```
>>> my_model.name

"csv_data"

>>> my_model.name = "cleaned_data"
>>> my_model.name

"cleaned_data"
```

27.8.3 *LogisticRegressionModel* predict

predict (*self*, *frame*, *observation_columns=None*)

[ALPHA] Make a new frame with a column for label prediction.

Parameters **frame** : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame whose labels are to be predicted. By default, predict is run on the same columns over which the model is trained.

observation_columns : list (default=None)

Column(s) containing the observations whose labels are to be predicted. By default, we predict the labels over columns the `LogisticRegressionModel` was trained on.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

Frame containing the original frame's columns and a column with the predicted label.

Predict the labels for a test frame and create a new frame revision with existing columns and a new predicted label's column.

27.8.4 *LogisticRegressionModel* test

test (*self*, *frame*, *label_column*, *observation_columns=None*)

[ALPHA] Predict test frame labels and show metrics.

Parameters **frame** : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
frame whose labels are to be predicted.

label_column : unicode
Column containing the actual label for each observation.

observation_columns : list (default=None)
Column(s) containing the observations whose labels are to be predicted and tested.
Default is to test over the columns the `SvmModel` was trained on.

Returns : dict

object An object with binary classification metrics. The data returned is composed of multiple components:

<object>.accuracy : double <object>.confusion_matrix : table <object>.f_measure :
double <object>.precision : double <object>.recall : double

Predict the labels for a test frame and run classification metrics on predicted and target labels.

27.8.5 *LogisticRegressionModel* train

train (*self*, *frame*, *label_column*, *observation_columns*, *frequency_column=None*, *num_classes=2*, *optimizer='LBFGS'*, *compute_covariance=True*, *intercept=True*, *feature_scaling=False*, *threshold=0.5*, *reg_type='L2'*, *reg_param=0.0*, *num_iterations=100*, *convergence_tolerance=0.0001*, *num_corrections=10*, *mini_batch_fraction=1.0*, *step_size=1*)
[ALPHA] Build logistic regression model.

Parameters **frame** : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
A frame to train the model on.

label_column : unicode
Column name containing the label for each observation.

observation_columns : list
Column(s) containing the observations.

frequency_column : unicode (default=None)
Optional column containing the frequency of observations.

num_classes : int32 (default=2)
Number of classes

optimizer : unicode (default=LBFGS)

Set type of optimizer. LBFGS - Limited-memory BFGS. LBFGS supports multinomial logistic regression. SGD - Stochastic Gradient Descent. SGD only supports binary logistic regression.

compute_covariance : bool (default=True)

If true, compute covariance matrix for the model.

intercept : bool (default=True)

If true, add intercept column to training data.

feature_scaling : bool (default=False)

If true, perform feature scaling before training model.

threshold : float64 (default=0.5)

Threshold for separating positive predictions from negative predictions.

reg_type : unicode (default=L2)

Set type of regularization L1 - L1 regularization with sum of absolute values of coefficients L2 - L2 regularization with sum of squares of coefficients

reg_param : float64 (default=0.0)

Regularization parameter

num_iterations : int32 (default=100)

Maximum number of iterations

convergence_tolerance : float64 (default=0.0001)

Convergence tolerance of iterations for L-BFGS. Smaller value will lead to higher accuracy with the cost of more iterations.

num_corrections : int32 (default=10)

Number of corrections used in LBFGS update. Default 10. Values of numCorrections less than 3 are not recommended; large values of numCorrections will result in excessive computing time.

mini_batch_fraction : float64 (default=1.0)

Fraction of data to be used for each SGD iteration

step_size : int32 (default=1)

Initial step size for SGD. In subsequent steps, the step size decreases by $\text{stepSize}/\sqrt{t}$

Returns : dict

object An object with a summary of the trained model. The data returned is composed of multiple components:

numFeatures [Int] Number of features in the training data

numClasses [Int] Number of classes in the training data

summaryTable: table A summary table composed of:

covarianceMatrix: Frame (optional) Covariance matrix of the trained model. The covariance matrix is the inverse of the Hessian matrix for the trained model. The Hessian matrix is the second-order partial derivatives of the model's log-likelihood function

“”

Creating a LogisticRegression Model using the observation column and label column of the train frame.

class LogisticRegressionModel

Create a ‘new’ instance of logistic regression model.

Attributes

<code>name</code>	Set or get the name of the model object.
-------------------	--

Methods

<code>__init__(self[, name, _info])</code>	Create a ‘new’ instance of logistic regression model.
<code>predict(self, frame[, observation_columns])</code>	[ALPHA] Make a new frame with a column for label prediction.
<code>test(self, frame, label_column[, observation_columns])</code>	[ALPHA] Predict test frame labels and show metrics.
<code>train(self, frame, label_column, observation_columns[, frequency_column, ...])</code>	[ALPHA] Build logistic regression model.

`__init__(self, name=None)`

Create a ‘new’ instance of logistic regression model.

Parameters `name` : unicode (default=None)

User supplied name.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

27.9 Models NaiveBayesModel

27.9.1 NaiveBayesModel new

`__init__(self, name=None)`

create a new model

Parameters `name` : unicode (default=None)

User supplied name.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

27.9.2 *NaiveBayesModel* name

name

Set or get the name of the model object.

Parameters

Change or retrieve model object identification. Identification names must start with a letter and are limited to alphanumeric characters and the `_` character.

Examples

```
>>> my_model.name

"csv_data"

>>> my_model.name = "cleaned_data"
>>> my_model.name

"cleaned_data"
```

27.9.3 *NaiveBayesModel* predict

predict (*self*, *frame*, *observation_columns=None*)
[ALPHA] Predict

Parameters **frame** : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame whose labels are to be predicted. By default, predict is run on the same columns over which the model is trained.

observation_columns : list (default=None)

Column(s) containing the observations whose labels are to be predicted. By default, we predict the labels over columns the `NaiveBayesModel` was trained on.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

27.9.4 *NaiveBayesModel* train

train (*self*, *frame*, *label_column*, *observation_columns*, *lambda_parameter=None*)
[ALPHA] Build a naive bayes model.

Parameters **frame** : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame to train the model on.

label_column : unicode

Column containing the label for each observation.

observation_columns : list

Column(s) containing the observations.

lambda_parameter : float64 (default=None)

Additive smoothing parameter Default is 1.0.

Returns : _Unit

Train a NaiveBayesModel using the observation column, label column of the train frame and an optional lambda value.

class NaiveBayesModel
create a new model

Attributes

name	Set or get the name of the model object.
-------------	--

Methods

<code>__init__(self[, name, _info])</code>	create a new model
<code>predict(self, frame[, observation_columns])</code>	[ALPHA] Predict
<code>train(self, frame, label_column, observation_columns[, lambda_parameter])</code>	[ALPHA] Build a naive bayes model.

`__init__(self, name=None)`
create a new model

Parameters **name** : unicode (default=None)

User supplied name.

Returns : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

27.10 Models LinearRegressionModel

27.10.1 LinearRegressionModel new

`__init__(self, name=None)`

Parameters **name** : unicode (default=None)

User supplied name.

Returns : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

27.10.2 *LinearRegressionModel* name

name

Set or get the name of the model object.

Parameters

Change or retrieve model object identification. Identification names must start with a letter and are limited to alphanumeric characters and the `_` character.

Examples

```
>>> my_model.name

"csv_data"

>>> my_model.name = "cleaned_data"
>>> my_model.name

"cleaned_data"
```

27.10.3 *LinearRegressionModel* predict

predict (*self*, *frame*, *observation_columns=None*)

[ALPHA] Make new frame with column for label prediction.

Parameters **frame** : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame whose labels are to be predicted. By default, predict is run on the same columns over which the model is trained.

observation_columns : list (default=None)

Column(s) containing the observations whose labels are to be predicted. By default, we predict the labels over columns the `LogisticRegressionModel` was trained on.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

Frame containing the original frame's columns and a column with the predicted label.

Predict the labels for a test frame and create a new frame revision with existing columns and a new predicted label's column.

27.10.4 *LinearRegressionModel* train

train (*self*, *frame*, *label_column*, *observation_columns*, *intercept=None*, *num_iterations=None*,
step_size=None, *reg_type=None*, *reg_param=None*, *mini_batch_fraction=None*)
[ALPHA] Build linear regression model.

Parameters **frame** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame to train the model on.

label_column : unicode

Column name containing the label for each observation.

observation_columns : list

Column(s) containing the observations.

intercept : bool (default=None)

The algorithm adds an intercept. Default is true.

num_iterations : int32 (default=None)

Number of iterations. Default is 100.

step_size : int32 (default=None)

Step size for optimizer. Default is 1.0.

reg_type : unicode (default=None)

Regularization L1 or L2. Default is L2.

reg_param : float64 (default=None)

Regularization parameter. Default is 0.01.

mini_batch_fraction : float64 (default=None)

Mini batch fraction parameter. Default is 1.0.

Returns : _Unit

Creating a LinearRegression Model using the observation column and label column of the train frame.

class **LinearRegressionModel**

Entity LinearRegressionModel

Attributes

name	Set or get the name of the model object.
-------------	--

Methods

__init__ (self[, name, _info])	
predict (self, frame[, observation_columns])	[ALPHA] Make new frame with column for label prediction.
train (self, frame, label_column, observation_columns[, intercept, ...])	[ALPHA] Build linear regression model.

__init__ (self, name=None)

Parameters **name** : unicode (default=None)

User supplied name.

Returns : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

27.11 *Models* RandomForestRegressorModel

27.11.1 *RandomForestRegressorModel* new

__init__ (*self*, *name=None*)

<Missing Doc>

Parameters **name** : unicode (default=None)

User supplied name.

Returns : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

27.11.2 *RandomForestRegressorModel* name

name

Set or get the name of the model object.

Parameters

Change or retrieve model object identification. Identification names must start with a letter and are limited to alphanumeric characters and the `_` character.

Examples

```
>>> my_model.name

"csv_data"

>>> my_model.name = "cleaned_data"
>>> my_model.name

"cleaned_data"
```

27.11.3 *RandomForestRegressorModel* predict

predict (*self*, *frame*, *observation_columns=None*)

[ALPHA] Predict the values for the data points.

Parameters **frame** : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame whose labels are to be predicted. By default, predict is run on the same columns over which the model is trained.

observation_columns : list (default=None)

Column(s) containing the observations whose labels are to be predicted. By default, we predict the labels over columns the RandomForestModel was trained on.

Returns : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

Frame A new frame consisting of the existing columns of the frame and a new column with predicted value for each observation.

27.11.4 *RandomForestRegressorModel* publish

publish (*self*)

[BETA] Creates a tar file that will be used as input to the scoring engine

Parameters

Returns : dict

Returns the HDFS path to the tar file

Creates a tar file with the trained Random Forest Regressor Model The tar file is used as input to the scoring engine to predict the value of an observation.

27.11.5 *RandomForestRegressorModel* train

train (*self*, *frame*, *label_column*, *observation_columns*, *num_trees=1*, *impurity='variance'*, *max_depth=4*, *max_bins=100*, *seed=284863502*, *categorical_features_info=None*, *feature_subset_category=None*)
[ALPHA] Build Random Forests Regressor model.

Parameters **frame** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame to train the model on

label_column : unicode

Column name containing the label for each observation

observation_columns : list

Column(s) containing the observations

num_trees : int32 (default=1)

Number of trees in the random forest

impurity : unicode (default=variance)

Criterion used for information gain calculation. Supported values “variance”

max_depth : int32 (default=4)

Maximum depth of the tree

max_bins : int32 (default=100)

Maximum number of bins used for splitting features

seed : int32 (default=284863502)

Random seed for bootstrapping and choosing feature subsets

categorical_features_info : None (default=None)

feature_subset_category : unicode (default=None)

Number of features to consider for splits at each node. Supported values “auto”, “all”, “sqrt”, “log2”, “onethird”

Returns : dict

Creating a Random Forests Regressor Model using the observation columns and label column.

class RandomForestRegressorModel

<Missing Doc>

Attributes

name	Set or get the name of the model object.
-------------	--

Methods

__init__ (self[, name, _info])	<Missing Doc>
predict (self, frame[, observation_columns])	[ALPHA] Predict the values for the data points.
publish (self)	[BETA] Creates a tar file that will be used as input to the scoring engine
train (self, frame, label_column, observation_columns[, num_trees, impurity, ...])	[ALPHA] Build Random Forests Regressor model.

__init__(self, name=None)

<Missing Doc>

Parameters **name** : unicode (default=None)

User supplied name.

Returns : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

27.12 *trustedanalytics* get_model

get_model(*identifier*)

Get handle to a model object.

Parameters **identifier** : str | int

Name of the model to get

Returns : Model

model object

27.13 *trustedanalytics* drop_models

drop_models (*items*)

Deletes the model on the server.

Parameters **items** : [str | model object | list [str | model objects]]

Either the name of the model object to delete or the model object itself

27.14 *trustedanalytics* get_model_names

get_model_names ()

Retrieve names for all the model objects on the server.

Returns : list

List of names

Global Methods

`get_model`

`drop_models`

`get_model_names`

Part VI

REST API

REST API COMMANDS

28.1 *Commands* Issue Command

Issue a command for execution.

28.1.1 POST /v1/commands

Request

Route

POST /v1/commands

Body

Name	Description
name	full name of the command
arguments	JSON object specifying the command arguments

Here's an example showing how to issue the “assign_sample” command on frame 16:

```
{
  "name": "frame/assign_sample",
  "arguments": {
    "sample_labels": [
      "train",
      "test",
      "validate"
    ],
    "frame": 16,
    "random_seed": null,
    "sample_percentages": [
      0.5,
      0.3,
      0.2
    ],
    "output_column": null
  }
}
```

Headers

Authorization: test_api_key_1
Content-type: application/json

Response

Status

200 OK

Body

Returns information about the command. See the Response Body for [Get Command](#). It is the same. Note that POSTing to 'commands' creates the command and issues it for execution and immediately returns. To determine the command progress and status, use [Get Command](#).

28.2 *Commands* Get Command

Gets information about a specific command.

28.2.1 GET /v1/commands/:id

Request

Route

GET /v1/commands/25

Body

(None)

Headers

Authorization: test_api_key_1 Content-type: application/json

Response

Status

200 OK

Body

Returns information about the command

Name	Description
id	command instance id (engine-assigned)
name	command name
correlation_id	correlation id
links	links to the command
arguments	the arguments that were passed to the command
progress	command execution progress progress: percentage complete tasks_info: info about each task in the command including number of retries
complete	boolean indicating if the command has finished
result	the return data from command completion. (field will not appear until 'complete' is true)

Example response body for command 18, 'assign_sample' on frame 16 which is in the middle of execution:

```
{
  "id": 18,
  "name": "frame/assign_sample",
  "correlation_id": "",
  "arguments": {
    "sample_labels": ["train", "test", "validate"],
    "frame": 16,
    "random_seed": null,
    "sample_percentages": [0.5, 0.3, 0.2],
    "output_column": null
  },
  "progress": [{
    "progress": 33.33000183105469,
    "tasks_info": {
      "retries": 0
    }
  }],
  "complete": false,
  "links": [{
    "rel": "self",
    "uri": "http://localhost:9099/v1/commands/18",
    "method": "GET"
  }]
}
```

Example response body for command 17, a 'load' on frame 16 which has completed:

```
{
  "id": 17,
  "name": "frame:/load",
  "correlation_id": "3d074058-54bd-4170-a8a7-2219e6e3a894",
  "arguments": {
    "source": {
      "source_type": "file",
      "parser": {
        "name": "builtin/line/separator",
        "arguments": {
          "separator": ",",
          "skip_rows": 0,
          "schema": {
            "columns": [{"c", "int32"}, {"number", "unicode"}]
          }
        }
      }
    },
    "data": null,
    "uri": "/join_left.csv"
  },
  "destination": 16,
  "progress": [{
    "progress": 100.0,
    "tasks_info": {
      "retries": 0
    }
  }]
}
```

```
    },
    "complete": true,
    "result": {
      "id": 16,
      "name": "super_frame",
      "schema": {
        "columns": [{
          "name": "c",
          "data_type": "int32",
          "index": 0
        }, {
          "name": "number",
          "data_type": "string",
          "index": 1
        }]
      },
      "status": 1,
      "created_on": "2015-05-15T14:58:23.369-07:00",
      "modified_on": "2015-05-15T14:58:35.272-07:00",
      "storage_format": "file/parquet",
      "storage_location": "hdfs://paulsimon.hf.trustedanalytics.com/user/atkuser/trustedanalytics/frame",
      "row_count": 3,
      "modified_by": 1,
      "materialized_on": "2015-05-15T14:58:32.611-07:00",
      "materialization_complete": "2015-05-15T14:58:35.258-07:00",
      "last_read_date": "2015-05-15T14:58:23.369-07:00",
      "uri": "frames/16",
      "entity_type": "frame:"
    },
    "links": [{
      "rel": "self",
      "uri": "http://localhost:9099/v1/commands/17",
      "method": "GET"
    }]
  }
}
```

Headers

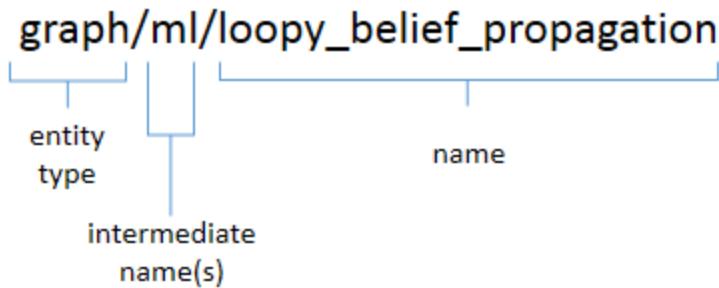
```
Content-Length: 405
Content-Type: application/json; charset=UTF-8
Date: Thu, 14 May 2015 23:42:27 GMT
```

Note

[Some notes could go here recommending polling strategies, and more info about the progress field]

28.3 *Commands* About Command Names

Command names are structured hierarchically according to entities and any intermediate scoping names. The full name of a command is delimited with the / character.



All commands are associated with an entity type, like a frame or a graph. The command name begins with the **entity type**. The `:` character indicates subtyping, where no `:` means all related entities inherit the command. Example entity types:

<code>frame</code>	# corresponds to all <code>*Frame</code> entity types
<code>frame:</code>	# indicates the standard <code>Frame</code> entity type
<code>frame:vertex</code>	# indicates the <code>VertexFrame</code> entity type
<code>model:kmeans</code>	# indicates the <code>KmeansModel</code> entity type

This means the command `frame/bin_columns` is available on any type of `Frame` object, where `frame:vertex/add_vertices` is only available on `VertexFrame` entities. The first argument to any command is the id of an entity instance. This entity instance must correspond to the supported entity type(s) indicated in the command's full name.

After the entity type, but before the name, there may be one or more intermediate names that provide additional scope.

Finally comes the name which identifies the operation.

28.4 Commands `_admin:/_explicit_garbage_collection`

<Missing Doc>

28.4.1 POST `/v1/commands/`

28.4.2 GET `/v1/commands/:id`

Request

Route

```
POST /v1/commands/
```

Body

name `_admin:/_explicit_garbage_collection`

arguments `age_to_delete_data` : unicode (default=None)

Minimum age of entity for data deletion. Defaults to server config.

`age_to_delete_meta_data` : unicode (default=None)

Minimum age of entity for meta data deletion. Defaults to server config.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.4.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
_Unit
```

28.5 *Commands* frame:/filter

Select all rows which satisfy a predicate.

28.5.1 POST /v1/commands/

28.5.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

Note - An argument for this command requires a Python User-Defined Function (UDF). This function must be especially prepared (wrapped/serialized) in order for it to run in the engine. **If this argument is needed for your call (i.e. it may be optional), then this particular command usage is NOT practically available as a REST API.** Today, the trustedanalytics Python client does the special function preparation and calls this API.

name frame:/filter

arguments frame : <bound method AtkEntityType.__name__ of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
 <Missing Description>

udf : None

<Missing Description>

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Modifies the current frame to save defined rows and delete everything else.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.5.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
_Unit
```

28.6 *Commands* frame:/join

[BETA] Join two data frames (similar to SQL JOIN).

28.6.1 POST /v1/commands/

28.6.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body**name** frame:/join**arguments left_frame** : None

<Missing Description>

right_frame : None

<Missing Description>

how : unicode

Methods of join (inner, left, right or outer).

name : unicode (default=None)

Name of new frame to be created.

skewed_join_type : unicode (default=None)

The type of skewed join: 'skewedhash' or 'skewedbroadcast'

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description**Response****Status**

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.6.3 GET /v1/commands/:id**Request****Route**

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
```

28.7 *Commands* frame:/label_propagation

Label Propagation on Gaussian Random Fields.

28.7.1 POST /v1/commands/

28.7.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame:/label_propagation

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

<Missing Description>

src_col_name : unicode

The column name for the source vertex id.

dest_col_name : unicode

The column name for the destination vertex id.

weight_col_name : unicode

The column name for the edge weight.

src_label_col_name : unicode

The column name for the label properties for the source vertex.

result_col_name : unicode (default=None)

The column name for the results (holding the post labels for the vertices).

max_iterations : int32 (default=None)

The maximum number of supersteps that the algorithm will execute. The valid value range is all positive int. Default is 10.

convergence_threshold : float32 (default=None)

The amount of change in cost function that will be tolerated at convergence. If the change is less than this threshold, the algorithm exits earlier before it reaches the maximum number of supersteps. The valid value range is all float and zero. Default is 0.00000001f.

alpha : float32 (default=None)

The tradeoff parameter that controls how much influence an external classifier's prediction contributes to the final prediction. This is for the case where an external classifier is available that can produce initial probabilistic classification on unlabeled examples, and the option allows incorporating external classifier's prediction into the LP training process. The valid value range is [0.0,1.0]. Default is 0.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Label Propagation on Gaussian Random Fields.

This algorithm is presented in X. Zhu and Z. Ghahramani. [Learning from labeled and unlabeled data with label propagation](http://www.cs.cmu.edu/~zhuxj/pub/CMU-CALD-02-107.pdf). Technical Report CMU-CALD-02-107, CMU, 2002¹.

Label Propagation (LP)

LP is a message passing technique for inputting or *smoothing* labels in partially-labelled datasets. Labels are propagated from *labeled* data to *unlabeled* data along a graph encoding similarity relationships among data points. The labels of known data can be probabilistic, in other words, a known point can be represented with fuzzy labels such as 90% label 0 and 10% label 1. The inverse distance between data points is represented by edge weights, with closer points having a higher weight (stronger influence on posterior estimates) than points farther away. LP has been used for many problems, particularly those involving a similarity measure between data points. Our implementation is based on Zhu and Ghahramani's 2002 paper, [Learning from labeled and unlabeled data](http://www.cs.cmu.edu/~zhuxj/pub/CMU-CALD-02-107.pdf).²

¹<http://www.cs.cmu.edu/~zhuxj/pub/CMU-CALD-02-107.pdf>

²<http://www.cs.cmu.edu/~zhuxj/pub/CMU-CALD-02-107.pdf>

The Label Propagation Algorithm

In LP, all nodes start with a prior distribution of states and the initial messages vertices pass to their neighbors are simply their prior beliefs. If certain observations have states that are known deterministically, they can be given a prior probability of 100% for their true state and 0% for all others. Unknown observations should be given uninformative priors.

Each node, i , receives messages from its k neighbors and updates its beliefs by taking a weighted average of its current beliefs and a weighted average of the messages received from its neighbors.

The updated beliefs for node i are:

$$updated\ belief_i = \lambda * (prior\ belief_i) + (1 - \lambda) * \sum_k w_{i,k} * previous\ belief_k$$

where $w_{i,k}$ is the normalized weight between nodes i and k , normalized such that the sum of all weights to neighbors is 1.

λ is a leaning parameter. If λ is greater than zero, updated probabilities will be anchored in the direction of prior beliefs.

The final distribution of state probabilities will also tend to be biased in the direction of the distribution of initial beliefs. For the first iteration of updates, nodes' previous beliefs are equal to the priors, and, in each future iteration, previous beliefs are equal to their beliefs as of the last iteration. All beliefs for every node will be updated in this fashion, including known observations, unless `anchor_threshold` is set. The `anchor_threshold` parameter specifies a probability threshold above which beliefs should no longer be updated. Hence, with an `anchor_threshold` of 0.99, observations with states known with 100% certainty will not be updated by this algorithm.

This process of updating and message passing continues until the convergence criteria is met, or the maximum number of *supersteps* is reached. A node is said to converge if the total change in its cost function is below the convergence threshold. The cost function for a node is given by:

$$cost = \sum_k w_{i,k} * \left[(1 - \lambda) * [previous\ belief_i^2 - w_{i,k} * previous\ belief_i * previous\ belief_k] + 0.5 * \lambda * (previous\ belief_i - prior_i)^2 \right]$$

Convergence is a local phenomenon; not all nodes will converge at the same time. It is also possible that some (most) nodes will converge and others will not converge. The algorithm requires all nodes to converge before declaring global convergence. If this condition is not met, the algorithm will continue up to the maximum number of *supersteps*.

Response

Status

200 OK

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.7.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

dict

A 2-column frame:

vertex: int A vertex id.

result [Vector (long)] label vector for the results (for the node id in column 1)

28.8 *Commands* frame:/load

Append data from a csv/xml into an existing (possibly empty) frame

28.8.1 POST /v1/commands/

28.8.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame:/load

arguments destination : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

<Missing Description>

source : None

<Missing Description>

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Append data from a csv/xml into an existing (possibly empty) frame

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.8.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

200 OK

Body

_Unit

28.9 *Commands* frame:/loopy_belief_propagation

Message passing to infer state probabilities.

28.9.1 POST /v1/commands/

28.9.2 GET /v1/commands/:id

Request

Route

POST /v1/commands/

Body

name frame:/loopy_belief_propagation

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

<Missing Description>

src_col_name : unicode

The column name for the source vertex id.

dest_col_name : unicode

The column name for the destination vertex id.

weight_col_name : unicode

The column name for the edge weight.

src_label_col_name : unicode

The column name for the label properties for the source vertex.

result_col_name : unicode (default=None)

The column name for the results (holding the post labels for the vertices).

ignore_vertex_type : bool (default=None)

If True, all vertex will be treated as training data. Default is False.

max_iterations : int32 (default=None)

The maximum number of supersteps that the algorithm will execute. The valid value range is all positive int. The default value is 10.

convergence_threshold : float32 (default=None)

The amount of change in cost function that will be tolerated at convergence. If the change is less than this threshold, the algorithm exits earlier before it reaches the maximum number of supersteps. The valid value range is all float and zero. The default value is 0.00000001f.

anchor_threshold : float64 (default=None)

The parameter that determines if a node's posterior will be updated or not. If a node's maximum prior value is greater than this threshold, the node will be treated as anchor node, whose posterior will inherit from prior without update. This is for the case where we have confident prior estimation for some nodes and don't want the algorithm to update these nodes. The valid value range is in [0, 1]. Default is 1.0.

smoothing : float32 (default=None)

The Ising smoothing parameter. This parameter adjusts the relative strength of closeness encoded edge weights, similar to the width of Gaussian distribution. Larger value implies smoother decay and the edge weight becomes less important. Default is 2.0.

max_product : bool (default=None)

Should LBP use max_product or not. Default is False.

power : float32 (default=None)

Power coefficient for power edge potential. Default is 0.

Headers

Authorization: test_api_key_1 Content-type: application/json

Description

Loopy belief propagation on *Markov Random Fields* (MRF). *Belief Propagation* (BP) was originally designed for acyclic graphical models, then it was found that the BP algorithm can be used in general graphs. The algorithm is then sometimes called “Loopy” Belief Propagation (LBP), because graphs typically contain cycles, or loops.

Loopy Belief Propagation (LBP)

Loopy Belief Propagation (LBP) is a message passing algorithm for inferring state probabilities, given a graph and a set of noisy initial estimates. The LBP implementation assumes that the joint distribution of the data is given by a Boltzmann distribution.

For more information about LBP, see: “K. Murphy, Y. Weiss, and M. Jordan, Loopy-belief Propagation for Approximate Inference: An Empirical Study, UAI 1999.”

LBP has a wide range of applications in structured prediction, such as low-level vision and influence spread in social networks, where we have prior noisy predictions for a large set of random variables and a graph encoding relationships between those variables.

The algorithm performs approximate inference on an *undirected graph* of hidden variables, where each variable is represented as a node, and each edge encodes relations to its neighbors. Initially, a prior noisy estimate of state probabilities is given to each node, then the algorithm infers the posterior distribution of each node by propagating and collecting messages to and from its neighbors and updating the beliefs.

In graphs containing loops, convergence is not guaranteed, though LBP has demonstrated empirical success in many areas and in practice often converges close to the true joint probability distribution.

Discrete Loopy Belief Propagation

LBP is typically considered a *semi-supervised machine learning* algorithm as

1. there is typically no ground truth observation of states
2. the algorithm is primarily concerned with estimating a joint probability function rather than with *classification* or point prediction.

The standard (discrete) LBP algorithm requires a set of probability thresholds to be considered a classifier. Nonetheless, the discrete LBP algorithm allows Test/Train/Validate splits of the data and the algorithm will treat “Train” observations differently from “Test” and “Validate” observations. Vertices labelled with “Test” or “Validate” will be treated as though they have uninformative (uniform) priors and are allowed to receive messages, but not send messages. This simulates a “scoring scenario” in which a new observation is added to a graph containing fully trained LBP posteriors, the new vertex is scored based on received messages, but the full LBP algorithm is not repeated in full. This behavior can be turned off by setting the `ignore_vertex_type` parameter to True. When `ignore_vertex_type=True`, all nodes will be considered “Train” regardless of their sample type designation. The Gaussian (continuous) version of LBP does not allow Train/Test/Validate splits.

The standard LBP algorithm included with the toolkit assumes an ordinal and cardinal set of discrete states. For notational convenience, we’ll denote the value of state s_i as i , and the prior probability of state s_i as $prior_i$.

Each node sends out initial messages of the form:

$$\ln \left(\sum_{s_j} \exp \left(-\frac{|i-j|^p}{n-1} * w * s + \ln(prior_i) \right) \right)$$

Where

- w is the weight between the messages destination and origin vertices
- s is the *smoothing* parameter
- p is the power parameter
- n is the number of states

The larger the weight between two nodes, or the higher the smoothing parameter, the more neighboring vertices are assumed to “agree” on states. We represent messages as sums of log probabilities rather than products of non-logged probabilities which makes it easier to subtract messages in the future steps of the algorithm. Also note that the states are cardinal in the sense that the “pull” of state i on state j depends on the distance between i and j . The *power* parameter intensifies the rate at which the pull of distant states drops off.

In order for the algorithm to work properly, all edges of the graph must be bidirectional. In other words, messages need to be able to flow in both directions across every edge. Bidirectional edges can be enforced during graph building, but the LBP function provides an option to do an initial check for bidirectionality using the `bidirectional_check=True` option. If not all the edges of the graph are bidirectional, the algorithm will return an error.

Look at a case where a node has two states, 0 and 1. The 0 state has a prior probability of 0.9 and the 1 state has a prior probability of 0.2. The states have uniform weights of 1, power of 1 and a smoothing parameter of 2. The nodes initial message would be $[\ln(0.2 + 0.8e^{-2}), \ln(0.8 + 0.2e^{-2})]$, which gets sent to each of that node’s neighbors. Note that messages will typically not be proper probability distributions, hence each message is normalized so that

the probability of all states sum to 1 before being sent out. For simplicity of discussion, we will consider all messages as normalized messages.

After nodes have sent out their initial messages, they then update their beliefs based on messages that they have received from their neighbors, denoted by the set k .

Updated Posterior Beliefs:

$$\ln(\text{newbelief}) = \exp \left[\ln(\text{prior}) + \sum_k \text{message}_k \right]$$

Note that the messages in the above equation are still in log form. Nodes then send out new messages which take the same form as their initial messages, with updated beliefs in place of priors and subtracting out the information previously received from the new message's recipient. The recipient's prior message is subtracted out to prevent feedback loops of nodes "learning" from themselves.

$$\ln \left(\sum_{s_j} \exp \left(-\frac{|i-j|^p}{n-1} * w * s + \ln(\text{newbelief}_i) - \text{previous message from recipient} \right) \right)$$

In updating beliefs, new beliefs tend to be most influenced by the largest message. Setting the `max_product` option to "True" ignores all incoming messages other than the strongest signal. Doing this results in approximate solutions, but requires significantly less memory and run-time than the more exact computation. Users should consider this option when processing power is a constraint and approximate solutions to LBP will be sufficient.

This process of updating and message passing continues until the convergence criteria is met or the maximum number of *supersteps* is reached without converging. A node is said to converge if the total change in its distribution (the sum of absolute value changes in state probabilities) is less than the `convergence_threshold` parameter. Convergence is a local phenomenon; not all nodes will converge at the same time. It is also possible for some (most) nodes to converge and others to never converge. The algorithm requires all nodes to converge before declaring that the algorithm has converged overall. If this condition is not met, the algorithm will continue up to the maximum number of *supersteps*.

See: http://en.wikipedia.org/wiki/Belief_propagation.

Response

Status

200 OK

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.9.3 GET /v1/commands/:id

Request

Route

GET /v1/commands/18

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response**Status**

```
200 OK
```

Body

```
dict
```

a 2-column frame:

vertex: int A vertex id.

result [Vector (long)] label vector for the results (for the node id in column 1).

28.10 *Commands* frame:/rename_columns

Rename columns

28.10.1 POST /v1/commands/

28.10.2 GET /v1/commands/:id

Request**Route**

```
POST /v1/commands/
```

Body

name frame:/rename_columns

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

<Missing Description>

names : None

<Missing Description>

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.10.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
_Unit
```

28.11 *Commands* frame:edge/add_edges

Add edges to a graph.

28.11.1 POST /v1/commands/

28.11.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame:edge/add_edges

arguments **edge_frame** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

source_frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
Frame that will be the source of the edge data.

column_name_for_source_vertex_id : unicode
column name for a unique id for each source vertex (this is not the system defined _vid).

column_name_for_dest_vertex_id : unicode
column name for a unique id for each destination vertex (this is not the system defined _vid).

column_names : list (default=None)
Column names to be used as properties for each vertex, None means use all columns,
empty list means use none.

create_missing_vertices : bool (default=False)
True to create missing vertices for edge (slightly slower), False to drop edges pointing to
missing vertices. Defaults to False.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Includes appending to a list of existing edges.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.11.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
_Unit
```

28.12 *Commands* frame:edge/rename_columns

Rename columns for edge frame.

28.12.1 POST /v1/commands/

28.12.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame:edge/rename_columns

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

names : None

<Missing Description>

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.12.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response**Status**

```
200 OK
```

Body

_Unit

28.13 *Commands* frame:vertex/add_vertices

Add vertices to a graph.

28.13.1 POST /v1/commands/

28.13.2 GET /v1/commands/:id

Request**Route**

```
POST /v1/commands/
```

Body

name frame:vertex/add_vertices

arguments **vertex_frame** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

<Missing Description>

source_frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

Frame that will be the source of the vertex data.

id_column_name : unicode

Column name for a unique id for each vertex.

column_names : list (default=None)

Column names that will be turned into properties for each vertex.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Includes appending to a list of existing vertices.

Response**Status**

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.13.3 GET /v1/commands/:id**Request****Route**

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

200 OK

Body

_Unit

28.14 *Commands* frame:vertex/drop_duplicates

Remove duplicate vertex rows.

28.14.1 POST /v1/commands/

28.14.2 GET /v1/commands/:id

Request

Route

POST /v1/commands/

Body

name frame:vertex/drop_duplicates

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

<Missing Description>

unique_columns : None (default=None)

<Missing Description>

Headers

Authorization: test_api_key_1
Content-type: application/json

Description

Remove duplicate vertex rows, keeping only one vertex row per uniqueness criteria match. Edges that were connected to removed vertices are also automatically dropped.

Response

Status

200 OK

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.14.3 GET /v1/commands/:id

Request

Route

GET /v1/commands/18

Body

(None)

Headers

Authorization: test_api_key_1 Content-type: application/json

Response

Status

200 OK

Body

_Unit

28.15 *Commands* frame:vertex/filter

28.15.1 POST /v1/commands/

28.15.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

Note - An argument for this command requires a Python User-Defined Function (UDF). This function must be especially prepared (wrapped/serialized) in order for it to run in the engine. **If this argument is needed for your call (i.e. it may be optional), then this particular command usage is NOT practically available as a REST API.** Today, the trustedanalytics Python client does the special function preparation and calls this API.

name frame:vertex/filter

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

udf : None

<Missing Description>

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.15.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response**Status**

```
200 OK
```

Body

```
_Unit
```

28.16 *Commands* frame:vertex/rename_columns

Rename columns for vertex frame.

28.16.1 POST /v1/commands/

28.16.2 GET /v1/commands/:id

Request**Route**

```
POST /v1/commands/
```

Body

name frame:vertex/rename_columns

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

<Missing Description>

names : None

<Missing Description>

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.16.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
_Unit
```

28.17 *Commands* frame/_coalesce

Calls underlying Spark RDD method.

28.17.1 POST /v1/commands/

28.17.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/_coalesce

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

number_partitions : int32
number of Spark RDD partitions

shuffle : bool (default=False)
shuffle data between partitions

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.17.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
_Unit
```

28.18 *Commands* frame/_partition_count

Calls underlying Spark RDD method.

28.18.1 POST /v1/commands/

28.18.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/_partition_count

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

<Missing Description>

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.18.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

200 OK

Body

dict

28.19 *Commands* frame/_repartition

Calls underlying Spark RDD method.

28.19.1 POST /v1/commands/

28.19.2 GET /v1/commands/:id

Request**Route**

POST /v1/commands/

Body

name frame/_repartition

arguments frame : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

number_partitions : int32
number of RDD partitions

Headers

Authorization: test_api_key_1
Content-type: application/json

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.19.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
_Unit
```

28.20 *Commands* frame/_size_on_disk

Calculate the size on disk in bytes of a frame.

28.20.1 POST /v1/commands/

28.20.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/_size_on_disk

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.20.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

200 OK

Body

dict

28.21 *Commands* frame/add_columns

Add columns to current frame.

28.21.1 POST /v1/commands/

28.21.2 GET /v1/commands/:id

Request

Route

POST /v1/commands/

Body

Note - An argument for this command requires a Python User-Defined Function (UDF). This function must be especially prepared (wrapped/serialized) in order for it to run in the engine. **If this argument is needed for your call (i.e. it may be optional), then this particular command usage is NOT practically available as a REST API.** Today, the trustedanalytics Python client does the special function preparation and calls this API.

name frame/add_columns

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

Frame to which new columns need to be added

column_names : list

List of names for the new columns

column_types : list

List of data types for the new columns

udf : None

<Missing Description>

columns_accessed : list

List of columns which the UDF will access. This adds significant performance benefit if we know which column(s) will be needed to execute the UDF, especially when the frame has significantly more columns than those being used to evaluate the UDF.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Assigns data to column based on evaluating a function for each row.

Notes

1. The row UDF ('func') must return a value in the same format as specified by the schema. See [Python User Functions](#).
2. Unicode in column names is not supported and will likely cause the `drop_frames()` method (and others) to fail!

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.21.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

200 OK

Body

_Unit

28.22 *Commands* frame/assign_sample

Randomly group rows into user-defined classes.

28.22.1 POST /v1/commands/

28.22.2 GET /v1/commands/:id

Request

Route

POST /v1/commands/

Body

name frame/assign_sample

arguments **frame** : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

<Missing Description>

sample_percentages : list

Entries are non-negative and sum to 1. (See the note below.) If the i 'th entry of the list is p , then then each row receives label i with independent probability p .

sample_labels : list (default=None)

Names to be used for the split classes. Defaults "TR", "TE", "VA" when the length of *sample_percentages* is 3, and defaults to Sample_0, Sample_1, ... otherwise.

output_column : unicode (default=None)

Name of the new column which holds the labels generated by the function.

random_seed : int32 (default=None)

Random seed used to generate the labels. Defaults to 0.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Randomly assign classes to rows given a vector of percentages. The table receives an additional column that contains a random label. The random label is generated by a probability distribution function. The distribution function is specified by the `sample_percentages`, a list of floating point values, which add up to 1. The labels are non-negative integers drawn from the range $[0, \text{len}(S) - 1]$ where S is the `sample_percentages`. Optionally, the user can specify a list of strings to be used as the labels. If the number of labels is 3, the labels will default to “TR”, “TE” and “VA”.

Notes

The sample percentages provided by the user are preserved to at least eight decimal places, but beyond this there may be small changes due to floating point imprecision.

In particular:

1. The engine validates that the sum of probabilities sums to 1.0 within eight decimal places and returns an error if the sum falls outside of this range.
2. The probability of the final class is clamped so that each row receives a valid label with probability one.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.22.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

200 OK

Body

_Unit

28.23 *Commands* frame/bin_column

Classify data into user-defined groups.

28.23.1 POST /v1/commands/

28.23.2 GET /v1/commands/:id

Request

Route

POST /v1/commands/

Body

name frame/bin_column

arguments **frame** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

<Missing Description>

column_name : unicode

Name of the column to bin.

cutoffs : list

Array of values containing bin cutoff points. Array can be list or tuple. Array values must be progressively increasing. All bin boundaries must be included, so, with N bins, you need N+1 values.

include_lowest : bool (default=None)

Specify how the boundary conditions are handled. True indicates that the lower bound of the bin is inclusive. False indicates that the upper bound is inclusive. Default is True.

strict_binning : bool (default=None)

Specify how values outside of the cutoffs array should be binned. If set to True, each value less than cutoffs[0] or greater than cutoffs[-1] will be assigned a bin value of -1. If set to False, values less than cutoffs[0] will be included in the first bin while values greater than cutoffs[-1] will be included in the final bin.

bin_column_name : unicode (default=None)

The name for the new binned column. Default is `<column_name>_binned`.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Summarize rows of data based on the value in a single column by sorting them into bins, or groups, based on a list of bin cutoff points.

Notes

1. Unicode in column names is not supported and will likely cause the `drop_frames()` method (and others) to fail!
2. Bins IDs are 0-index: the lowest bin number is 0.
3. The first and last cutoffs are always included in the bins. When `include_lowest` is `True`, the last bin includes both cutoffs. When `include_lowest` is `False`, the first bin (bin 0) includes both cutoffs.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.23.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```


Response

Status

200 OK

Body

_Unit

28.24 *Commands* frame/bin_column_equal_depth

Classify column into groups with the same frequency.

28.24.1 POST /v1/commands/

28.24.2 GET /v1/commands/:id

Request

Route

POST /v1/commands/

Body

name frame/bin_column_equal_depth

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

Identifier for the input dataframe.

column_name : unicode

The column whose values are to be binned.

num_bins : int32 (default=None)

The maximum number of bins. Default is the Square-root choice $\lfloor \sqrt{m} \rfloor$, where m is the number of rows.

bin_column_name : unicode (default=None)

The name for the new column holding the grouping labels. Default is
<column_name>_binned.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Group rows of data based on the value in a single column and add a label to identify grouping.

Equal depth binning attempts to label rows such that each bin contains the same number of elements. For n bins of a column C of length m , the bin number is determined by:

$$\lceil n * \frac{f(C)}{m} \rceil$$

where f is a tie-adjusted ranking function over values of C . If there are multiples of the same value in C , then their tie-adjusted rank is the average of their ordered rank values.

Notes

1. Unicode in column names is not supported and will likely cause the `drop_frames()` method (and others) to fail!
2. The `num_bins` parameter is considered to be the maximum permissible number of bins because the data may dictate fewer bins. For example, if the column to be binned has a quantity of X elements with only 2 distinct values and the `num_bins` parameter is greater than 2, then the actual number of bins will only be 2. This is due to a restriction that elements with an identical value must belong to the same bin.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.24.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
dict
```

A list containing the edges of each bin.

28.25 *Commands* frame/bin_column_equal_width

Classify column into same-width groups.

28.25.1 POST /v1/commands/

28.25.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/bin_column_equal_width

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

Identifier for the input dataframe.

column_name : unicode

The column whose values are to be binned.

num_bins : int32 (default=None)

The maximum number of bins. Default is the Square-root choice $\lfloor \sqrt{m} \rfloor$, where m is the number of rows.

bin_column_name : unicode (default=None)

The name for the new column holding the grouping labels. Default is
<column_name>_binned.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Group rows of data based on the value in a single column and add a label to identify grouping.

Equal width binning places column values into groups such that the values in each group fall within the same interval and the interval width for each group is equal.

Notes

1. Unicode in column names is not supported and will likely cause the `drop_frames()` method (and others) to fail!
2. The `num_bins` parameter is considered to be the maximum permissible number of bins because the data may dictate fewer bins. For example, if the column to be binned has 10 elements with only 2 distinct values and the `num_bins` parameter is greater than 2, then the number of actual number of bins will only be 2. This is due to a restriction that elements with an identical value must belong to the same bin.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.25.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
dict
```

A list of the edges of each bin.

28.26 *Commands* frame/categorical_summary

Build summary of the data.

28.26.1 POST /v1/commands/

28.26.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/categorical_summary

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

column_input : list

List of Categorical Column Input consisting of column, topk and/or threshold

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Optional parameters:

top_k : *int* Displays levels which are in the top k most frequently occurring values for that column.
Default is 10.

threshold : *float* Displays levels which are above the threshold percentage with respect to the total row count. Default is 0.0.

Compute a summary of the data in a column(s) for categorical or numerical data types. The returned value is a Map containing categorical summary for each specified column.

For each column, levels which satisfy the top k and/or threshold cutoffs are displayed along with their frequency and percentage occurrence with respect to the total rows in the dataset.

Performs level pruning first based on top k and then filters out levels which satisfy the threshold criterion.

Missing data is reported when a column value is empty (“”) or null.

All remaining data is grouped together in the Other category and its frequency and percentage are reported as well.

User must specify the column name and can optionally specify top_k and/or threshold.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.26.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

200 OK

Body

dict

Summary for specified column(s) consisting of levels with their frequency and percentage.

28.27 *Commands* frame/classification_metrics

Model statistics of accuracy, precision, and others.

28.27.1 POST /v1/commands/

28.27.2 GET /v1/commands/:id

Request

Route

POST /v1/commands/

Body

name frame/classification_metrics

arguments frame : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

label_column : unicode

The name of the column containing the correct label for each instance.

pred_column : unicode

The name of the column containing the predicted label for each instance.

pos_label : None (default=None)

<Missing Description>

beta : float64 (default=None)

This is the beta value to use for F_β measure (default F1 measure is computed); must be greater than zero. Defaults is 1.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Calculate the accuracy, precision, confusion_matrix, recall and F_β measure for a classification model.

- The **f_measure** result is the F_β measure for a classification model. The F_β measure of a binary classification model is the harmonic mean of precision and recall. If we let:

- $\beta \equiv \beta$,
- T_P denotes the number of true positives,
- F_P denotes the number of false positives, and
- F_N denotes the number of false negatives

then:

$$F_\beta = (1 + \beta^2) * \frac{\frac{T_P}{T_P + F_P} * \frac{T_P}{T_P + F_N}}{\beta^2 * \frac{T_P}{T_P + F_P} + \frac{T_P}{T_P + F_N}}$$

The F_β measure for a multi-class classification model is computed as the weighted average of the F_β measure for each label, where the weight is the number of instances of each label. The determination of binary vs. multi-class is automatically inferred from the data.

- The **recall** result of a binary classification model is the proportion of positive instances that are correctly identified. If we let T_P denote the number of true positives and F_N denote the number of false negatives, then the model recall is given by $\frac{T_P}{T_P + F_N}$.

For multi-class classification models, the recall measure is computed as the weighted average of the recall for each label, where the weight is the number of instances of each label. The determination of binary vs. multi-class is automatically inferred from the data.

- The **precision** of a binary classification model is the proportion of predicted positive instances that are correctly identified. If we let T_P denote the number of true positives and F_P denote the number of false positives, then the model precision is given by: $\frac{T_P}{T_P + F_P}$.

For multi-class classification models, the precision measure is computed as the weighted average of the precision for each label, where the weight is the number of instances of each label. The determination of binary vs. multi-class is automatically inferred from the data.

- The **accuracy** of a classification model is the proportion of predictions that are correctly identified. If we let T_P denote the number of true positives, T_N denote the number of true negatives, and K denote the total number of classified instances, then the model accuracy is given by: $\frac{T_P + T_N}{K}$.

This measure applies to binary and multi-class classifiers.

- The **confusion_matrix** result is a confusion matrix for a binary classifier model, formatted for human readability.

Notes

The **confusion_matrix** is not yet implemented for multi-class classifiers.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.27.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
dict
```

```
object <object>.accuracy : double <object>.confusion_matrix : table <object>.f_measure : double
<object>.precision : double <object>.recall : double
```

28.28 *Commands* frame/column_median

Calculate the (weighted) median of a column.

28.28.1 POST /v1/commands/

28.28.2 GET /v1/commands/:id

Request

Route

POST /v1/commands/

Body

name frame/column_median

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

data_column : unicode

The column whose median is to be calculated.

weights_column : unicode (default=None)

The column that provides weights (frequencies) for the median calculation. Must contain numerical data. Default is all items have a weight of 1.

Headers

Authorization: test_api_key_1 Content-type: application/json

Description

The median is the least value X in the range of the distribution so that the cumulative weight of values strictly below X is strictly less than half of the total weight and the cumulative weight of values up to and including X is greater than or equal to one-half of the total weight.

All data elements of weight less than or equal to 0 are excluded from the calculation, as are all data elements whose weight is NaN or infinite. If a weight column is provided and no weights are finite numbers greater than 0, None is returned.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.28.3 GET /v1/commands/:id**Request****Route**

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response**Status**

```
200 OK
```

Body

```
dict
```

varies The median of the values. If a weight column is provided and no weights are finite numbers greater than 0, None is returned. The type of the median returned is the same as the contents of the data column, so a column of Longs will result in a Long median and a column of Floats will result in a Float median.

28.29 Commands frame/column_mode

Evaluate the weights assigned to rows.

28.29.1 POST /v1/commands/**28.29.2 GET /v1/commands/:id****Request****Route**

```
POST /v1/commands/
```

Body

name frame/column_mode

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

data_column : unicode

Name of the column supplying the data.

weights_column : unicode (default=None)

Name of the column supplying the weights. Default is all items have weight of 1.

max_modes_returned : int32 (default=None)

Maximum number of modes returned. Default is 1.

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Calculate the modes of a column. A mode is a data element of maximum weight. All data elements of weight less than or equal to 0 are excluded from the calculation, as are all data elements whose weight is NaN or infinite. If there are no data elements of finite weight greater than 0, no mode is returned.

Because data distributions often have multiple modes, it is possible for a set of modes to be returned. By default, only one is returned, but by setting the optional parameter `max_modes_returned`, a larger number of modes can be returned.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.29.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

dict

dict Dictionary containing summary statistics. The data returned is composed of multiple components:

mode [A mode is a data element of maximum net weight.] A set of modes is returned. The empty set is returned when the sum of the weights is 0. If the number of modes is less than or equal to the parameter `max_modes_returned`, then all modes of the data are returned. If the number of modes is greater than the `max_modes_returned` parameter, only the first `max_modes_returned` many modes (per a canonical ordering) are returned.

weight_of_mode [Weight of a mode.] If there are no data elements of finite weight greater than 0, the weight of the mode is 0. If no weights column is given, this is the number of appearances of each mode.

total_weight [Sum of all weights in the weight column.] This is the row count if no weights are given. If no weights column is given, this is the number of rows in the table with non-zero weight.

mode_count [The number of distinct modes in the data.] In the case that the data is very multimodal, this number may exceed `max_modes_returned`.

28.30 *Commands* frame/column_summary_statistics

Calculate multiple statistics for a column.

28.30.1 POST /v1/commands/

28.30.2 GET /v1/commands/:id

Request

Route

POST /v1/commands/

Body

name frame/column_summary_statistics

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

data_column : unicode

The column to be statistically summarized. Must contain numerical data; all NaNs and infinite values are excluded from the calculation.

weights_column : unicode (default=None)

Name of column holding weights of column values.

use_population_variance : bool (default=None)

If true, the variance is calculated as the population variance. If false, the variance calculated as the sample variance. Because this option affects the variance, it affects the standard deviation and the confidence intervals as well. Default is false.

Headers

Authorization: test_api_key_1
Content-type: application/json

Description

Notes

Sample Variance Sample Variance is computed by the following formula:

$$\left(\frac{1}{W-1}\right) * \sum_i (x_i - M)^2$$

where W is sum of weights over valid elements of positive weight, and M is the weighted mean.

Population Variance Population Variance is computed by the following formula:

$$\left(\frac{1}{W}\right) * \sum_i (x_i - M)^2$$

where W is sum of weights over valid elements of positive weight, and M is the weighted mean.

Standard Deviation The square root of the variance.

Logging Invalid Data A row is bad when it contains a NaN or infinite value in either its data or weights column. In this case, it contributes to `bad_row_count`; otherwise it contributes to good row count.

A good row can be skipped because the value in its weight column is less than or equal to 0. In this case, it contributes to `non_positive_weight_count`, otherwise (when the weight is greater than 0) it contributes to `valid_data_weight_pair_count`.

Equations `bad_row_count + good_row_count = # rows in the frame`

`positive_weight_count + non_positive_weight_count = good_row_count`

In particular, when no weights column is provided and all weights are 1.0,

`non_positive_weight_count = 0` and `positive_weight_count = good_row_count`

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.30.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

200 OK

Body

dict

dict Dictionary containing summary statistics. The data returned is composed of multiple components:

mean [[double | None]] Arithmetic mean of the data.

geometric_mean [[double | None]] Geometric mean of the data. None when there is a data element ≤ 0 , 1.0 when there are no data elements.

variance [[double | None]] None when there are ≤ 1 many data elements. Sample variance is the weighted sum of the squared distance of each data element from the weighted mean, divided by the total weight minus 1. None when the sum of the weights is ≤ 1 . Population variance is the weighted sum of the squared distance of each data element from the weighted mean, divided by the total weight.

standard_deviation [[double | None]] The square root of the variance. None when sample variance is being used and the sum of weights is ≤ 1 .

total_weight [long] The count of all data elements that are finite numbers. (In other words, after excluding NaNs and infinite values.)

minimum [[double | None]] Minimum value in the data. None when there are no data elements.

maximum [[double | None]] Maximum value in the data. None when there are no data elements.

mean_confidence_lower [[double | None]] Lower limit of the 95% confidence interval about the mean. Assumes a Gaussian distribution. None when there are no elements of positive weight.

mean_confidence_upper [[double | None]] Upper limit of the 95% confidence interval about the mean. Assumes a Gaussian distribution. None when there are no elements of positive weight.

bad_row_count [[double | None]] The number of rows containing a NaN or infinite value in either the data or weights column.

good_row_count [[double | None]] The number of rows not containing a NaN or infinite value in either the data or weights column.

positive_weight_count [[double | None]] The number of valid data elements with weight > 0 . This is the number of entries used in the statistical calculation.

non_positive_weight_count [[double | None]] The number valid data elements with finite weight ≤ 0 .

28.31 *Commands* frame/compute_misplaced_score

28.31.1 POST /v1/commands/

28.31.2 GET /v1/commands/:id

Request

Route


```
POST /v1/commands/
```

Body

name frame/compute_misplaced_score

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

gravity : float64

Similarity measure for computing tension between 2 connected items

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description**Response****Status**

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.31.3 GET /v1/commands/:id**Request****Route**

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
```

28.32 *Commands* frame/copy

New frame with copied columns.

28.32.1 POST /v1/commands/

28.32.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

Note - An argument for this command requires a Python User-Defined Function (UDF). This function must be especially prepared (wrapped/serialized) in order for it to run in the engine. **If this argument is needed for your call (i.e. it may be optional), then this particular command usage is NOT practically available as a REST API.** Today, the trustedanalytics Python client does the special function preparation and calls this API.

name frame/copy

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

<Missing Description>

columns : None (default=None)

<Missing Description>

where : None (default=None)

<Missing Description>

name : unicode (default=None)

name of the frame copy

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Copies specified columns into a new Frame object, optionally renaming them and/or filtering them.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.32.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

200 OK

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

New Frame object.
```

28.33 *Commands* frame/correlation

Calculate correlation for two columns of current frame.

28.33.1 POST /v1/commands/

28.33.2 GET /v1/commands/:id

Request

Route

POST /v1/commands/

Body

name frame/correlation

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

data_column_names : list

The names of 2 columns from which to compute the correlation.

Headers

Authorization: test_api_key_1
Content-type: application/json

Description

Notes

This method applies only to columns containing numerical data.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.33.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
dict
```

Pearson correlation coefficient of the two columns.

28.34 *Commands* frame/correlation_matrix

Calculate correlation matrix for two or more columns.

28.34.1 POST /v1/commands/

28.34.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/correlation_matrix

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

data_column_names : list

The names of the columns from which to compute the matrix.

matrix_name : unicode (default=None)

The name for the returned matrix Frame.

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Notes

This method applies only to columns containing numerical data.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.34.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
```

A Frame with the matrix of the correlation values for the columns.

28.35 *Commands frame/count_where*

Counts qualified rows.

28.35.1 POST /v1/commands/

28.35.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

Note - An argument for this command requires a Python User-Defined Function (UDF). This function must be especially prepared (wrapped/serialized) in order for it to run in the engine. **If this argument is needed for your call (i.e. it may be optional), then this particular command usage is NOT practically available as a REST API.** Today, the trustedanalytics Python client does the special function preparation and calls this API.

name frame/count_where

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

<Missing Description>

udf : None

<Missing Description>

Headers

Authorization: test_api_key_1 Content-type: application/json

Description

Counts rows which meet criteria specified by a UDF predicate.

Response

Status

200 OK

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.35.3 GET /v1/commands/:id

Request

Route

GET /v1/commands/18

Body

(None)

Headers

Authorization: test_api_key_1 Content-type: application/json

Response

Status

```
200 OK
```

Body

```
dict
```

Number of rows matching qualifications.

28.36 *Commands* frame/covariance

Calculate covariance for exactly two columns.

28.36.1 POST /v1/commands/

28.36.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/covariance

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

data_column_names : list

The names of two columns from which to compute the covariance.

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Notes

This method applies only to columns containing numerical data.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.36.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
dict
```

Covariance of the two columns.

28.37 *Commands* frame/covariance_matrix

Calculate covariance matrix for two or more columns.

28.37.1 POST /v1/commands/

28.37.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/covariance_matrix

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

data_column_names : list

The names of the column from which to compute the matrix. Names should refer to a single column of type vector, or two or more columns of numeric scalars.

matrix_name : unicode (default=None)

The name of the new matrix.

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Notes

This function applies only to columns containing numerical data.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.37.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
```

A matrix with the covariance values for the columns.

28.38 *Commands* frame/cumulative_percent

[BETA] Add column to frame with cumulative percent sum.

28.38.1 POST /v1/commands/

28.38.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/cumulative_percent

arguments frame : <bound method AtkEntityType.__name__ of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

<Missing Description>

sample_col : unicode

The name of the column from which to compute the cumulative percent sum.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

A cumulative percent sum is computed by sequentially stepping through the rows, observing the column values and keeping track of the current percentage of the total sum accounted for at the current value.

Notes

This method applies only to columns containing numerical data. Although this method will execute for columns containing negative values, the interpretation of the result will change (for example, negative percentages).

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.38.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
_Unit
```

28.39 *Commands* frame/cumulative_sum

[BETA] Add column to frame with cumulative percent sum.

28.39.1 POST /v1/commands/

28.39.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/cumulative_sum

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

Identifier for the input dataframe

sample_col : unicode

The name of the column from which to compute the cumulative sum.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

A cumulative sum is computed by sequentially stepping through the rows, observing the column values and keeping track of the cumulative sum for each value.

Notes

This method applies only to columns containing numerical data.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.39.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

200 OK

Body

_Unit

28.40 *Commands* frame/dot_product

[ALPHA] Calculate dot product for each row in current frame.

28.40.1 POST /v1/commands/

28.40.2 GET /v1/commands/:id

Request

Route

POST /v1/commands/

Body

name frame/dot_product

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

<Missing Description>

left_column_names : list

Names of columns used to create the left vector (A) for each row. Names should refer to a single column of type vector, or two or more columns of numeric scalars.

right_column_names : list

Names of columns used to create right vector (B) for each row. Names should refer to a single column of type vector, or two or more columns of numeric scalars.

dot_product_column_name : unicode

Name of column used to store the dot product.

default_left_values : list (default=None)

Default values used to substitute null values in left vector. Default is None.

default_right_values : list (default=None)

Default values used to substitute null values in right vector. Default is None.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Calculate the dot product for each row in a frame using values from two equal-length sequences of columns.

Dot product is computed by the following formula:

The dot product of two vectors $A = [a_1, a_2, \dots, a_n]$ and $B = [b_1, b_2, \dots, b_n]$ is $a_1 * b_1 + a_2 * b_2 + \dots + a_n * b_n$. The dot product for each row is stored in a new column in the existing frame.

Notes

If `default_left_values` or `default_right_values` are not specified, any null values will be replaced by zeros.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.40.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
_Unit
```

28.41 *Commands* frame/drop_columns

Remove columns from the frame.

28.41.1 POST /v1/commands/

28.41.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/drop_columns

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

columns : list

Column name OR list of column names to be removed from the frame.

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

The data from the columns is lost.

Notes

It is not possible to delete all columns from a frame. At least one column needs to remain. If it is necessary to delete all columns, then delete the frame.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.41.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
_Unit
```

28.42 *Commands* frame/drop_duplicates

Modify the current frame, removing duplicate rows.

28.42.1 POST /v1/commands/

28.42.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/drop_duplicates

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

unique_columns : None (default=None)
<Missing Description>

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Remove data rows which are the same as other rows. The entire row can be checked for duplication, or the search for duplicates can be limited to one or more columns. This modifies the current frame.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.42.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
_Unit
```

28.43 *Commands* frame/ecdf

Builds new frame with columns for data and distribution.

28.43.1 POST /v1/commands/

28.43.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/ecdf

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

<Missing Description>

column : unicode

The name of the input column containing sample.

result_frame_name : unicode (default=None)

A name for the resulting frame which is created by this operation.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Generates the *empirical cumulative distribution* for the input column.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.43.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
```

A new Frame containing each distinct value in the sample and its corresponding ECDF value.

28.44 *Commands* frame/entropy

Calculate the Shannon entropy of a column.

28.44.1 POST /v1/commands/

28.44.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/entropy

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

<Missing Description>

data_column : unicode

The column whose entropy is to be calculated.

weights_column : unicode (default=None)

The column that provides weights (frequencies) for the entropy calculation. Must contain numerical data. Default is using uniform weights of 1 for all items.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

The data column is weighted via the weights column. All data elements of weight ≤ 0 are excluded from the calculation, as are all data elements whose weight is NaN or infinite. If there are no data elements with a finite weight greater than 0, the entropy is zero.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.44.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
dict
```

Entropy.

28.45 *Commands* frame/export_to_csv

Write current frame to HDFS in csv format.

28.45.1 POST /v1/commands/

28.45.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/export_to_csv

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

folder_name : unicode

The HDFS folder path where the files will be created.

separator : None (default=None)

<Missing Description>

count : int32 (default=None)

The number of records you want. Default, or a non-positive value, is the whole frame.

offset : int32 (default=None)

The number of rows to skip before exporting to the file. Default is zero (0).

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Export the frame to a file in csv format as a Hadoop file.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.45.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
_Unit
```

28.46 *Commands* frame/export_to_hbase

Write current frame to HBase table.

28.46.1 POST /v1/commands/

28.46.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/export_to_hbase

arguments frame : <bound method AtkEntityType.__name__ of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

Frame being exported to HBase

table_name : unicode

The name of the HBase table that will contain the exported frame

key_column_name : unicode (default=None)

The name of the column to be used as row key in hbase table

family_name : unicode (default=None)

The family name of the HBase table that will contain the exported frame

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Table must exist in HBase. Export of Vectors is not currently supported.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.46.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
_Unit
```

28.47 *Commands* frame/export_to_hive

Write current frame to Hive table.

28.47.1 POST /v1/commands/

28.47.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/export_to_hive

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

table_name : unicode

The name of the Hive table that will contain the exported frame

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Table must not exist in Hive. Export of Vectors is not currently supported.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.47.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
_Unit
```

28.48 *Commands* frame/export_to_jdbc

Write current frame to Jdbc table.

28.48.1 POST /v1/commands/

28.48.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/export_to_jdbc

arguments frame : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

Frame to be exported to jdbc

table_name : unicode

jdbc table name

connector_type : unicode (default=None)

(optional) jdbc connector type

url : unicode (default=None)

(optional) connection url (includes server name, database name, user acct and password)

driver_name : unicode (default=None)

(optional) driver name

query : unicode (default=None)

(optional) query for filtering. Not supported yet.

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Table will be created or appended to. Export of Vectors is not currently supported.

Response

Status

200 OK

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.48.3 GET /v1/commands/:id

Request

Route

GET /v1/commands/18

Body

(None)

Headers

Authorization: test_api_key_1 Content-type: application/json

Response

Status

200 OK

Body

_Unit

28.49 *Commands* frame/export_to_json

Write current frame to HDFS in JSON format.

28.49.1 POST /v1/commands/

28.49.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/export_to_json

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

folder_name : unicode

The HDFS folder path where the files will be created.

count : int32 (default=None)

The number of records you want. Default, or a non-positive value, is the whole frame.

offset : int32 (default=None)

The number of rows to skip before exporting to the file. Default is zero (0).

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Export the frame to a file in JSON format as a Hadoop file.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.49.3 GET /v1/commands/:id

Request

Route


```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response**Status**

```
200 OK
```

Body

```
_Unit
```

28.50 *Commands* frame/flatten_column

Spread data to multiple rows based on cell data.

28.50.1 POST /v1/commands/

28.50.2 GET /v1/commands/:id

Request**Route**

```
POST /v1/commands/
```

Body

name frame/flatten_column

arguments **frame** : <bound method `AtkEntityType.__name__` of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
 <Missing Description>

column : unicode

The column to be flattened.

delimiter : unicode (default=None)

The delimiter string. Default is comma (,).

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Splits cells in the specified column into multiple rows according to a string delimiter. New rows are a full copy of the original row, but the specified column only contains one value. The original row is deleted.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.50.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
_Unit
```

28.51 *Commands* frame/group_by

[BETA] Summarized Frame with Aggregations.

28.51.1 POST /v1/commands/

28.51.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/group_by

arguments frame : <bound method AtkEntityType.__name__ of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
 <Missing Description>

group_by_columns : list
 list of columns to group on

aggregations : list
 the aggregations to perform

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Create a Summarized Frame with Aggregations (Avg, Count, Max, Min, Mean, Sum, Stdev, ...).

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.51.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of  
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>  
Summarized Frame.
```

28.52 *Commands* frame/histogram

[BETA] Compute the histogram for a column in a frame.

28.52.1 POST /v1/commands/

28.52.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/histogram

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

<Missing Description>

column_name : unicode

Name of column to be evaluated.

num_bins : int32 (default=None)

Number of bins in histogram. Default is Square-root choice will be used (in other words `math.floor(math.sqrt(frame.row_count))`).

weight_column_name : unicode (default=None)

Name of column containing weights. Default is all observations are weighted equally.

bin_type : unicode (default=equalwidth)

The type of binning algorithm to use: ["equalwidth"|"equaldepth"] Defaults is "equalwidth".

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Compute the histogram of the data in a column. The returned value is a Histogram object containing 3 lists one each for: the cutoff points of the bins, size of each bin, and density of each bin.

Notes

The `num_bins` parameter is considered to be the maximum permissible number of bins because the data may dictate fewer bins. With equal depth binning, for example, if the column to be binned has 10 elements with only 2 distinct values and the `num_bins` parameter is greater than 2, then the number of actual number of bins will only be 2. This is due to a restriction that elements with an identical value must belong to the same bin.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.52.3 GET /v1/commands/:id**Request****Route**

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response**Status**

```
200 OK
```

Body

dict

histogram A Histogram object containing the result set. The data returned is composed of multiple components:

cutoffs [array of float] A list containing the edges of each bin.

hist [array of float] A list containing count of the weighted observations found in each bin.

density [array of float] A list containing a decimal containing the percentage of observations found in the total set per bin.

28.53 Commands frame/loadhbase

Append data from an hBase table into an existing (possibly empty) FrameRDD

28.53.1 POST /v1/commands/**28.53.2 GET /v1/commands/:id****Request****Route**

```
POST /v1/commands/
```

Body**name** frame/loadhbase

arguments destination : <bound method AtkEntityType.__name__ of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
 <Missing Description>

table_name : unicode

hbase table name

schema : list

hbase schema as a list of tuples (columnFamily, columnName, dataType for cell value)

start_tag : unicode (default=None)

optional start tag for filtering

end_tag : unicode (default=None)

optional end tag for filtering

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Append data from an hBase table into an existing (possibly empty) FrameRDD

Response**Status**

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.53.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
    the initial FrameRDD with the hbase data appended
```

28.54 *Commands* frame/loadhive

Append data from a hive table into an existing (possibly empty) frame

28.54.1 POST /v1/commands/

28.54.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/loadhive

arguments destination : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
DataFrame to load data into.Should be either a uri or id.

query : unicode

Initial query to run at load time

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Append data from a hive table into an existing (possibly empty) frame

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.54.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

200 OK

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

the initial frame with the hive data appended
```

28.55 *Commands* frame/loadjdbc

Append data from a Jdbc table into an existing (possibly empty) frame

28.55.1 POST /v1/commands/

28.55.2 GET /v1/commands/:id

Request

Route

POST /v1/commands/

Body

name frame/loadjdbc

arguments destination : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

DataFrame to load data into. Should be either a uri or id.

table_name : unicode

table name

connector_type : unicode (default=None)

(optional) connector type

url : unicode (default=None)

(optional) connection url (includes server name, database name, user acct and password)

driver_name : unicode (default=None)

(optional) driver name

query : unicode (default=None)

(optional) query for filtering. Not supported yet.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Append data from a Jdbc table into an existing (possibly empty) frame

Response**Status**

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.55.3 GET /v1/commands/:id**Request****Route**

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response**Status**

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
```

the initial frame with the Jdbc data appended

28.56 *Commands* frame/quantiles

New frame with Quantiles and their values.

28.56.1 POST /v1/commands/

28.56.2 GET /v1/commands/:id

Request

Route

POST /v1/commands/

Body

name frame/quantiles

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

column_name : unicode

The column to calculate quantiles.

quantiles : list

What is being requested.

Headers

Authorization: test_api_key_1 Content-type: application/json

Description

Calculate quantiles on the given column.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.56.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of  
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
```

A new frame with two columns (float64): requested Quantiles and their respective values.

28.57 *Commands* frame/rename

Change the name of the current frame.

28.57.1 POST /v1/commands/

28.57.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/rename

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

new_name : unicode

The new name of the frame.

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Set the name of this frame.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.57.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
_Unit
```

28.58 *Commands* frame/sort

[BETA] Sort by one or more columns.

28.58.1 POST /v1/commands/

28.58.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/sort

arguments frame : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

column_names_and_ascending : list

Column names to sort by, true for ascending, false for descending.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.58.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
_Unit
```

28.59 *Commands* frame/sorted_k

[ALPHA] Get a sorted subset of the data.

28.59.1 POST /v1/commands/

28.59.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/sorted_k

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

k : int32

Number of sorted records to return.

column_names_and_ascending : list

Column names to sort by, and true to sort column by ascending order, or false for descending order.

reduce_tree_depth : int32 (default=None)

Advanced tuning parameter which determines the depth of the reduce-tree for the sorted_k plugin. This plugin uses Spark's treeReduce() for scalability. The default depth is 2.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Take the first k (sorted) rows for the currently active Frame. Rows are sorted by column values in either ascending or descending order.

Returning the first k (sorted) rows is more efficient than sorting the entire frame when k is much smaller than the number of rows in the frame.

Notes

The number of sorted rows (k) should be much smaller than the number of rows in the original frame.

In particular:

1. The number of sorted rows (k) returned should fit in Spark driver memory.

The maximum size of serialized results that can fit in the Spark driver is set by the Spark configuration parameter *spark.driver.maxResultSize*.

2. If you encounter a Kryo buffer overflow exception, increase the Spark configuration parameter *spark.kryoserializer.buffer.max.mb*.
3. Use `Frame.sort()` instead if the number of sorted rows (k) is very large (i.e., cannot fit in Spark driver memory).

Response

Status

200 OK

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.59.3 GET /v1/commands/:id

Request

Route

GET /v1/commands/18

Body

(None)

Headers

Authorization: test_api_key_1 Content-type: application/json

Response

Status

200 OK

Body

```
<bound method AtkEntityType.__name__ of  
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
```

A new frame with the first k sorted rows from the original frame.

28.60 *Commands* frame/tally

[BETA] Count number of times a value is seen.

28.60.1 POST /v1/commands/

28.60.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/tally

arguments frame : <bound method `AtkEntityType.__name__` of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
 <Missing Description>

sample_col : unicode

The name of the column from which to compute the cumulative count.

count_val : unicode

The column value to be used for the counts.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

A cumulative count is computed by sequentially stepping through the rows, observing the column values and keeping track of the the number of times the specified *count_value* has been seen.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.60.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
_Unit
```

28.61 *Commands* frame/tally_percent

[BETA] Compute a cumulative percent count.

28.61.1 POST /v1/commands/

28.61.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/tally_percent

arguments frame : <bound method AtkEntityType.__name__ of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
 <Missing Description>

sample_col : unicode

The name of the column from which to compute the cumulative sum.

count_val : unicode

The column value to be used for the counts.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

A cumulative percent count is computed by sequentially stepping through the rows, observing the column values and keeping track of the percentage of the total number of times the specified *count_value* has been seen up to the current value.

Response**Status**

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.61.3 GET /v1/commands/:id**Request****Route**

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
_Unit
```

28.62 *Commands* frame/top_k

Most or least frequent column values.

28.62.1 POST /v1/commands/

28.62.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/top_k

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

<Missing Description>

column_name : unicode

The column whose top (or bottom) K distinct values are to be calculated.

k : int32

Number of entries to return (If k is negative, return bottom k).

weights_column : unicode (default=None)

The column that provides weights (frequencies) for the topK calculation. Must contain numerical data. Default is 1 for all items.

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Calculate the top (or bottom) K distinct values by count of a column. The column can be weighted. All data elements of weight ≤ 0 are excluded from the calculation, as are all data elements whose weight is NaN or infinite. If there are no data elements of finite weight > 0 , then topK is empty.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.62.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
```

An object with access to the frame of data.

28.63 *Commands* frame/unflatten_column

Compacts data from multiple rows based on cell data.

28.63.1 POST /v1/commands/

28.63.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name frame/unflatten_column

arguments frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

<Missing Description>

composite_key_column_names : list

name of the user column to be used as keys for unflattening.

delimiter : unicode (default=None)

separator for the data in the result columns. Default is comma (,).

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```


Description

Groups together cells in all columns (less the composite key) using “,” as string delimiter. The original rows are deleted. The grouping takes place based on a composite key passed as arguments.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.63.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
_Unit
```

28.64 *Commands* graph:/_info

Get debug info about a graph.

28.64.1 POST /v1/commands/

28.64.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name graph:/_info

arguments graph : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f66757730d0>>
<Missing Description>

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.64.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response**Status**

200 OK

Body

dict

28.65 *Commands* graph:/define_edge_type

Define an edge type.

28.65.1 POST /v1/commands/

28.65.2 GET /v1/commands/:id

Request**Route**

```
POST /v1/commands/
```

Body

name graph:/define_edge_type

arguments **graph_ref** : <bound method AtkEntityType.__name__ of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f66757730d0>>
 <Missing Description>

label : unicode

Label of the edge type.

src_vertex_label : unicode

The src “type” of vertices this edge connects.

dest_vertex_label : unicode

The destination “type” of vertices this edge connects.

directed : bool (default=False)

True if edges are directed, false if they are undirected.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.65.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

200 OK

Body

_Unit

28.66 *Commands* graph:/define_vertex_type

Define a vertex type by label.

28.66.1 POST /v1/commands/

28.66.2 GET /v1/commands/:id

Request**Route**

POST /v1/commands/

Body

name graph:/define_vertex_type

arguments **graph_ref** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f66757730d0>>
<Missing Description>

label : unicode

Label of the vertex type.

Headers

Authorization: test_api_key_1
Content-type: application/json

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.66.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
_Unit
```

28.67 *Commands* graph:/edge_count

Get the total number of edges in the graph.

28.67.1 POST /v1/commands/

28.67.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name graph:/edge_count

arguments graph : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f66757730d0>>
<Missing Description>

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description**Response****Status**

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.67.3 GET /v1/commands/:id**Request****Route**

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
dict
```

28.68 *Commands* graph:/export_to_titan

Convert current graph to TitanGraph.

28.68.1 POST /v1/commands/

28.68.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name graph:/export_to_titan

arguments **graph** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f66757730d0>>
<Missing Description>

new_graph_name : unicode (default=None)

The name of the new graph. Default is None.

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Convert this Graph into a TitanGraph object. This will be a new graph backed by Titan with all of the data found in this graph.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.68.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
dict
```

A new TitanGraph.

28.69 *Commands* graph:/ml/kclique_percolation

[ALPHA] Find groups of vertices with similar attributes.

28.69.1 POST /v1/commands/

28.69.2 GET /v1/commands/:id

Request

Route

POST /v1/commands/

Body

name graph:/ml/kclique_percolation

arguments graph : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f66757730d0>>
<Missing Description>

clique_size : int32

The sizes of the cliques used to form communities. Larger values of clique size result in fewer, smaller communities that are more connected. Must be at least 2.

community_property_label : unicode

Name of the community property of vertex that will be updated/created in the graph. This property will contain for each vertex the set of communities that contain that vertex.

Headers

Authorization: test_api_key_1 Content-type: application/json

Description

Community Detection Using the K-Clique Percolation Algorithm

Overview

Modeling data as a graph captures relations, for example, friendship ties between social network users or chemical interactions between proteins. Analyzing the structure of the graph reveals collections (often termed ‘communities’) of vertices that are more likely to interact amongst each other. Examples could include a community of friends in a social network or a collection of highly interacting proteins in a cellular process.

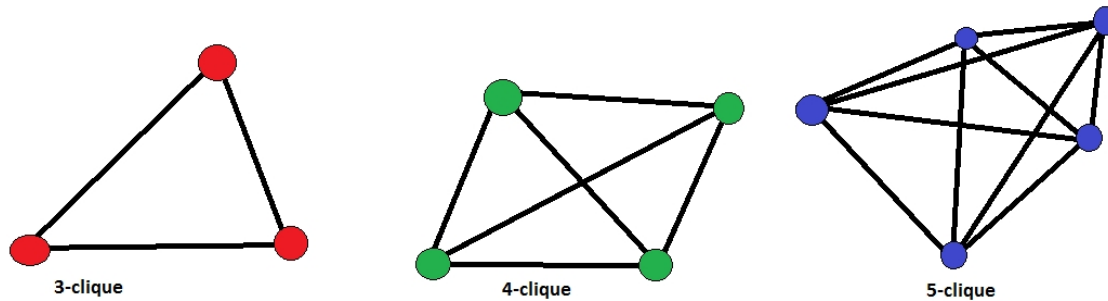
Trusted Analytics provides community detection using the k-Clique percolation method first proposed by Palla et. al.³ that has been widely used in many contexts.

K-Clique Percolation

³ G. Palla, I. Derenyi, I. Farkas, and T. Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. Nature, 435:814, 2005 (See <http://hal.elte.hu/cfinder/wiki/papers/communitylettm.pdf>)

K-clique percolation is a method for detecting community structure in graphs. Here we provide mathematical background on how communities are defined in the context of the k-clique percolation algorithm.

A clique is a group of vertices in which every vertex is connected (via undirected edge) with every other vertex in the clique. This graphically looks like a triangle or a structure composed of triangles:

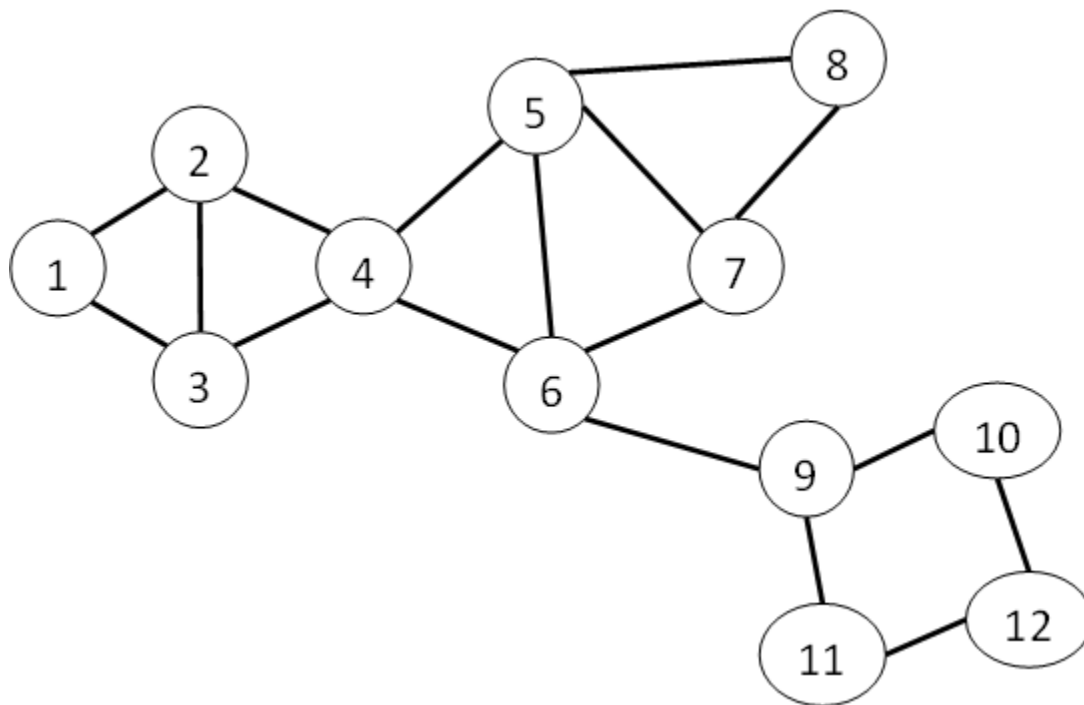


A clique is certainly a community in the sense that its vertices are all connected, but, it is too restrictive for most purposes, since it is natural some members of a community may not interact.

Mathematically, a k-clique has k vertices, each with $k - 1$ common edges, each of which connects to another vertex in the k-clique. The k-clique percolation method forms communities by taking unions of k-cliques that have $k - 1$ vertices in common.

K-Clique Example

In the graph below, the cliques are the sections defined by their triangular appearance and the 3-clique communities are $\{1, 2, 3, 4\}$ and $\{4, 5, 6, 7, 8\}$. The vertices 9, 10, 11, 12 are not in 3-cliques, therefore they do not belong to any community. Vertex 4 belongs to two distinct (but overlapping) communities.



Distributed Implementation of K-Clique Community Detection

The implementation of k-clique community detection in Trusted Analytics is a fully distributed implementation that follows the map-reduce algorithm proposed in Varamesh et. al.⁴.

⁴ Varamesh, A.; Akbari, M.K.; Fereiduni, M.; Sharifian, S.; Bagheri, A., "Distributed Clique Percolation based community detection on social

It has the following steps:

1. All k-cliques are *enumerated*.
2. k-cliques are used to build a “clique graph” by declaring each k-clique to be a vertex in a new graph and placing edges between k-cliques that share k-1 vertices in the base graph.
3. A *connected component* analysis is performed on the clique graph. Connected components of the clique graph correspond to k-clique communities in the base graph.
4. The connected components information for the clique graph is projected back down to the base graph, providing each vertex with the set of k-clique communities to which it belongs.

Notes

Spawns a number of Spark jobs that cannot be calculated before execution (it is bounded by the diameter of the clique graph derived from the input graph). For this reason, the initial loading, clique enumeration and clique-graph construction steps are tracked with a single progress bar (this is most of the time), and then successive iterations of analysis of the clique graph are tracked with many short-lived progress bars, and then finally the result is written out.

Response

Status

200 OK

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.69.3 GET /v1/commands/:id

Request

Route

GET /v1/commands/18

Body

(None)

Headers

Authorization: test_api_key_1 Content-type: application/json

networks using MapReduce,” Information and Knowledge Technology (IKT), 2013 5th Conference on, vol., no., pp.478,483, 28-30 May 2013

Response

Status

```
200 OK
```

Body

```
dict
```

Dictionary of vertex label and frame, Execution time.

28.70 *Commands* graph:/vertex_count

Get the total number of vertices in the graph.

28.70.1 POST /v1/commands/

28.70.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name graph:/vertex_count

arguments graph : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f66757730d0>>
<Missing Description>

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.70.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
dict
```

28.71 *Commands* graph:titan/export_to_graph

Export from ta.TitanGraph to ta.Graph.

28.71.1 POST /v1/commands/

28.71.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name graph:titan/export_to_graph

arguments graph : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f66757730d0>>
<Missing Description>

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description**Response****Status**

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.71.3 GET /v1/commands/:id**Request****Route**

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

200 OK

Body

dict

28.72 *Commands* graph:titan/graph_clustering

Performs graph clustering over an initial titan graph.

28.72.1 POST /v1/commands/

28.72.2 GET /v1/commands/:id

Request

Route

POST /v1/commands/

Body

name graph:titan/graph_clustering

arguments **graph** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f66757730d0>>
<Missing Description>

edge_distance : unicode

Column name for the edge distance.

Headers

Authorization: test_api_key_1 Content-type: application/json

Description

Performs graph clustering over an initial titan graph using a distributed edge collapse algorithm.

Response

Status

200 OK

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.72.3 GET /v1/commands/:id

Request

Route

GET /v1/commands/18

Body

(None)

Headers

Authorization: test_api_key_1 Content-type: application/json

Response

Status

200 OK

Body

_Unit

28.73 *Commands* graph:titan/query/gremlin

Executes a Gremlin query.

28.73.1 POST /v1/commands/

28.73.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name graph:titan/query/gremlin

arguments **graph** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f66757730d0>>
<Missing Description>

gremlin : unicode

The Gremlin script to execute.

Examples of Gremlin queries:

g.V[0..9] - Returns the first 10 vertices in graph
g.V.userId - Returns the userId property from vertices
g.V('name','hercules').out('father').out('father').name - Returns the name of Hercules' grandfather

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Executes a Gremlin query on an existing graph.

Notes

The query does not support pagination so the results of query should be limited using the Gremlin range filter [i..j], for example, g.V[0..9] to return the first 10 vertices.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.73.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

dict

List of query results serialized to JSON and runtime of Gremlin query in seconds. The list of results is in GraphSON format(for vertices or edges) or JSON (for other results like counts). GraphSON is a JSON-based format for property graphs which uses reserved keys that begin with underscores to encode vertex and edge metadata.

Examples of valid GraphSON:

```
{ \"name\": \"lop\", \"lang\": \"java\", \"_id\": \"3\", \"_type\": \"vertex\" }
{ \"weight\": 1, \"_id\": \"8\", \"_type\": \"edge\", \"_outV\": \"1\", \"_inV\": \"4\", \"_label\": \"weight\" }
```

See <https://github.com/tinkerpop/blueprints/wiki/GraphSON-Reader-and-Writer-Library>

28.74 *Commands* graph:titan/vertex_sample

Make subgraph from vertex sampling.

28.74.1 POST /v1/commands/

28.74.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name graph:titan/vertex_sample

arguments graph : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f66757730d0>>
<Missing Description>

size : int32

The number of vertices to sample from the graph.

sample_type : unicode

The type of vertex sample among: ['uniform', 'degree', 'degreedist'].

seed : int64 (default=None)

Random seed value.

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Create a vertex induced subgraph obtained by vertex sampling. Three types of vertex sampling are provided: 'uniform', 'degree', and 'degreedist'. A 'uniform' vertex sample is obtained by sampling vertices uniformly at random. For 'degree' vertex sampling, each vertex is weighted by its out-degree. For 'degreedist' vertex sampling, each vertex is weighted by the total number of vertices that have the same out-degree as it. That is, the weight applied to each vertex for 'degreedist' vertex sampling is given by the out-degree histogram bin size.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.74.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

dict

A new Graph object representing the vertex induced subgraph.

28.75 *Commands* graph/annotate_degrees

Make new graph with degrees.

28.75.1 POST /v1/commands/

28.75.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name graph/annotate_degrees

arguments **graph** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f66757730d0>>
<Missing Description>

output_property_name : unicode

The name of the new property. The degree is stored in this property.

degree_option : unicode (default=None)

Indicator for the definition of degree to be used for the calculation. Permitted values:

- “out” (default value) : Degree is calculated as the out-degree.
- “in” : Degree is calculated as the in-degree.
- “undirected” : Degree is calculated as the undirected degree. (Assumes that the edges are all undirected.)

Any prefix of the strings “out”, “in”, “undirected” will select the corresponding option.

input_edge_labels : list (default=None)

If this list is provided, only edges whose labels are included in the given set will be considered in the degree calculation. In the default situation (when no list is provided), all edges will be used in the degree calculation, regardless of label.

Headers

Authorization: test_api_key_1 Content-type: application/json

Description

Creates a new graph which is the same as the input graph, with the addition that every vertex of the graph has its *degree* stored in a user-specified property.

Degree Calculation

A fundamental quantity in graph analyses is the degree of a vertex: The degree of a vertex is the number of edges adjacent to it.

For a directed edge relation, a vertex has both an out-degree (the number of edges leaving the vertex) and an in-degree (the number of edges entering the vertex).

The toolkit provides this routine for calculating the degrees of vertices. This calculation could be performed with a Gremlin query on smaller datasets because Gremlin queries cannot be executed on a distributed scale. The Trusted Analytics routine `annotate_degrees` can be executed at distributed scale.

In the presence of edge weights, vertices can have weighted degrees: The weighted degree of a vertex is the sum of weights of edges adjacent to it. Analogously, the weighted in-degree of a vertex is the sum of the weights of the edges entering it, and the weighted out-degree is the sum of the weights of the edges leaving the vertex.

The toolkit provides `annotate_weighted_degrees` for the distributed calculation of weighted vertex degrees.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.75.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
dict
```

Dictionary containing the vertex type as the key and the corresponding vertex's frame with a column storing the annotated degree for the vertex in a user specified property. Call `dictionary_name['label']` to get the handle to frame whose vertex type is label.

28.76 *Commands* graph/annotate_weighted_degrees

Calculates the weighted degree of each vertex with respect to an (optional) set of labels.

28.76.1 POST /v1/commands/

28.76.2 GET /v1/commands/:id

Request

Route

POST /v1/commands/

Body

name graph/annotate_weighted_degrees

arguments graph : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f66757730d0>>
<Missing Description>

output_property_name : unicode
property name of where to store output

degree_option : unicode (default=None)
choose from 'out', 'in', 'undirected'

input_edge_labels : list (default=None)
labels of edge types that should be included

edge_weight_property : unicode (default=None)
property name of edge weight, if not provided all edges are weighted equally

edge_weight_default : float64 (default=None)
default edge weight

Headers

Authorization: test_api_key_1 Content-type: application/json

Description

Pulls graph from underlying store, calculates weighted degrees and writes them into the property specified, and then writes the output graph to the underlying store.

Degree Calculation

A fundamental quantity in graph analyses is the degree of a vertex: The degree of a vertex is the number of edges adjacent to it.

For a directed edge relation, a vertex has both an out-degree (the number of edges leaving the vertex) and an in-degree (the number of edges entering the vertex).

The toolkit provides a routine `annotate_degrees` for calculating the degrees of vertices. This calculation could be performed with a Gremlin query on smaller datasets because Gremlin queries cannot be executed on a distributed scale. The Trusted Analytics routine `annotate_degrees` can be executed at distributed scale.

In the presence of edge weights, vertices can have weighted degrees: The weighted degree of a vertex is the sum of weights of edges adjacent to it. Analogously, the weighted in-degree of a vertex is the sum of the weights of the edges entering it, and the weighted out-degree is the sum of the weights of the edges leaving the vertex.

The toolkit provides this routine for the distributed calculation of weighted vertex degrees.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.76.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
dict
```

28.77 *Commands* graph/clustering_coefficient

Coefficient of graph with respect to labels.

28.77.1 POST /v1/commands/

28.77.2 GET /v1/commands/:id

Request

Route

POST /v1/commands/

Body

name graph/clustering_coefficient

arguments **graph** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f66757730d0>>
<Missing Description>

output_property_name : unicode (default=None)

The name of the new property to which each vertex's local clustering coefficient will be written. If this option is not specified, no output frame will be produced and only the global clustering coefficient will be returned.

input_edge_labels : list (default=None)

If this list is provided, only edges whose labels are included in the given set will be considered in the clustering coefficient calculation. In the default situation (when no list is provided), all edges will be used in the calculation, regardless of label. It is required that all edges that enter into the clustering coefficient analysis be undirected.

Headers

Authorization: test_api_key_1 Content-type: application/json

Description

Calculates the clustering coefficient of the graph with respect to an (optional) set of labels.

Pulls graph from underlying store, calculates degrees and writes them into the property specified, and then writes the output graph to the underlying store.

Warning: THIS FUNCTION IS FOR UNDIRECTED GRAPHS. If it is called on a directed graph, its output is NOT guaranteed to calculate the local directed clustering coefficients.

Clustering Coefficients

The clustering coefficient of a graph provides a measure of how tightly clustered an undirected graph is. Informally, if the edge relation denotes “friendship”, the clustering coefficient of the graph is the probability that two people are friends given that they share a common friend.

More formally:

$$cc(G) = \frac{\|\{(u, v, w) \in V^3 : \{u, v\}, \{u, w\}, \{v, w\} \in E\}\|}{\|\{(u, v, w) \in V^3 : \{u, v\}, \{u, w\} \in E\}\|}$$

Analogously, the clustering coefficient of a vertex provides a measure of how tightly clustered that vertex’s neighborhood is. Informally, if the edge relation denotes “friendship”, the clustering coefficient at a vertex v is the probability that two acquaintances of v are themselves friends.

More formally:

$$cc(v) = \frac{\|\{(u, v, w) \in V^3 : \{u, v\}, \{u, w\}, \{v, w\} \in E\}\|}{\|\{(u, v, w) \in V^3 : \{v, u\}, \{v, w\} \in E\}\|}$$

The toolkit provides the function `clustering_coefficient` which computes both local and global clustering coefficients for a given undirected graph.

For more details on the mathematics and applications of clustering coefficients, see http://en.wikipedia.org/wiki/Clustering_coefficient.

Response

Status

200 OK

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.77.3 GET /v1/commands/:id

Request

Route

GET /v1/commands/18

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response**Status**

```
200 OK
```

Body

```
dict
```

Dictionary of the global clustering coefficient of the graph or, if local clustering coefficients are requested, a reference to the frame with local clustering coefficients stored at properties at each vertex.

28.78 *Commands graph/copy*

Make a copy of the current graph.

28.78.1 POST /v1/commands/

28.78.2 GET /v1/commands/:id

Request**Route**

```
POST /v1/commands/
```

Body

Note - An argument for this command requires a Python User-Defined Function (UDF). This function must be especially prepared (wrapped/serialized) in order for it to run in the engine. **If this argument is needed for your call (i.e. it may be optional), then this particular command usage is NOT practically available as a REST API.** Today, the trustedanalytics Python client does the special function preparation and calls this API.

name graph/copy

arguments **graph** : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f66757730d0>>
<Missing Description>

name : unicode (default=None)

The name for the copy of the graph. Default is None.

Headers

Authorization: test_api_key_1 Content-type: application/json

Description**Response****Status**

200 OK

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.78.3 GET /v1/commands/:id**Request****Route**

GET /v1/commands/18

Body

(None)

Headers

Authorization: test_api_key_1 Content-type: application/json

Response**Status**

200 OK

Body

dict

A copy of the original graph.

28.79 *Commands* graph/graphx_connected_components

Implements the connected components computation on a graph by invoking graphx api.

28.79.1 POST /v1/commands/

28.79.2 GET /v1/commands/:id

Request**Route**

POST /v1/commands/

Body

name graph/graphx_connected_components

arguments graph : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f66757730d0>>
<Missing Description>

output_property : unicode

The name of the column containing the connected component value.

Headers

Authorization: test_api_key_1 Content-type: application/json

Description

Pulls graph from underlying store, sends it off to the ConnectedComponentGraphXDefault, and then writes the output graph back to the underlying store.

Connected Components (CC)

Connected components are disjoint subgraphs in which all vertices are connected to all other vertices in the same component via paths, but not connected via paths to vertices in any other component. The connected components algorithm uses message passing along a specified edge type to find all of the connected components of a graph and label each edge with the identity of the component to which it belongs. The algorithm is specific to an edge type, hence in graphs with several different types of edges, there may be multiple, overlapping sets of connected components.

The algorithm works by assigning each vertex a unique numerical index and passing messages between neighbors. Vertices pass their indices back and forth with their neighbors and update their own index as the minimum of their current index and all other indices received. This algorithm continues until there is no change in any of the vertex indices. At the end of the algorithm, the unique levels of the indices denote the distinct connected components. The complexity of the algorithm is proportional to the diameter of the graph.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.79.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

dict

Dictionary containing the vertex type as the key and the corresponding vertex's frame with a connected component column. Call `dictionary_name['label']` to get the handle to frame whose vertex type is label.

28.80 *Commands* graph/graphx_pagerank

Determine which vertices are the most important.

28.80.1 POST /v1/commands/

28.80.2 GET /v1/commands/:id

Request**Route**

POST /v1/commands/

Body

name graph/graphx_pagerank

arguments **graph** : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f66757730d0>>
<Missing Description>

output_property : unicode

Name of the property to which pagerank value will be stored on vertex and edge.

input_edge_labels : list (default=None)

List of edge labels to consider for pagerank computation. Default is all edges are considered.

max_iterations : int32 (default=None)

The maximum number of iterations that will be invoked. The valid range is all positive int. Invalid value will terminate with vertex page rank set to `reset_probability`. Default is 20.

reset_probability : float64 (default=None)

The probability that the random walk of a page is reset. Default is 0.15.

convergence_tolerance : float64 (default=None)

The amount of change in cost function that will be tolerated at convergence. If this parameter is specified, `max_iterations` is not considered as a stopping condition. If the change is less than this threshold, the algorithm exits earlier. The valid value range is all float and zero. Default is 0.001.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Pulls graph from underlying store, sends it off to the PageRankRunner, and then writes the output graph back to the underlying store.

This method (currently) only supports Titan for graph storage.

**** Experimental Feature ****

Basics and Background

PageRank is a method for determining which vertices in a directed graph are the most central or important. *PageRank* gives each vertex a score which can be interpreted as the probability that a person randomly walking along the edges of the graph will visit that vertex.

The calculation of *PageRank* is based on the supposition that if a vertex has many vertices pointing to it, then it is “important”, and that a vertex grows in importance as more important vertices point to it. The calculation is based only on the network structure of the graph and makes no use of any side data, properties, user-provided scores or similar non-topological information.

PageRank was most famously used as the core of the Google search engine for many years, but as a general measure of *centrality* in a graph, it has other uses to other problems, such as *recommendation systems* and analyzing predator-prey food webs to predict extinctions.

Background references

- Basic description and principles: [Wikipedia: PageRank](#)⁵
- Applications to food web analysis: [Stanford: Applications of PageRank](#)⁶
- Applications to recommendation systems: [PLOS: Computational Biology](#)⁷

Mathematical Details of PageRank Implementation

The Trusted Analytics implementation of *PageRank* satisfies the following equation at each vertex v of the graph:

$$PR(v) = \frac{\rho}{n} + \rho \left(\sum_{u \in InSet(v)} \frac{PR(u)}{L(u)} \right)$$

Where:

- v — a vertex
- $L(v)$ — outbound degree of the vertex v
- $PR(v)$ — *PageRank* score of the vertex v
- $InSet(v)$ — set of vertices pointing to the vertex v
- n — total number of vertices in the graph
- ρ — user specified damping factor (also known as reset probability)

⁵<http://en.wikipedia.org/wiki/PageRank>

⁶<http://web.stanford.edu/class/msande233/handouts/lecture8.pdf>

⁷<http://www.ploscompbiol.org/article/fetchObject.action?uri=info%3Adoi%2F10.1371%2Fjournal.pcbi.1000494&representation=PDF>

Termination is guaranteed by two mechanisms.

- The user can specify a convergence threshold so that the algorithm will terminate when, at every vertex, the difference between successive approximations to the *PageRank* score falls below the convergence threshold.
- The user can specify a maximum number of iterations after which the algorithm will terminate.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.80.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

dict

dict((vertex_dictionary, (label, Frame)), (edge_dictionary, (label, Frame))).

Dictionary containing dictionaries of labeled vertices and labeled edges.

For the *vertex_dictionary* the vertex type is the key and the corresponding vertex's frame with a new column storing the page rank value for the vertex. Call `vertex_dictionary['label']` to get the handle to frame whose vertex type is label.

For the *edge_dictionary* the edge type is the key and the corresponding edge's frame with a new column storing the page rank value for the edge. Call `edge_dictionary['label']` to get the handle to frame whose edge type is label.

28.81 *Commands* graph/graphx_triangle_count

Number of triangles among vertices of current graph.

28.81.1 POST /v1/commands/

28.81.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name graph/graphx_triangle_count

arguments **graph** : <bound method `AtkEntityType.__name__` of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f66757730d0>>
<Missing Description>

output_property : unicode

The name of output property to be added to vertex/edge upon completion.

input_edge_labels : list (default=None)

The name of edge labels to be considered for triangle count. Default is all edges are considered.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

**** Experimental Feature ****

Counts the number of triangles among vertices in an undirected graph. If an edge is marked bidirectional, the implementation opts for canonical orientation of edges hence counting it only once (similar to an undirected graph).

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.81.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

dict

dict(label, Frame).

Dictionary containing the vertex type as the key and the corresponding vertex's frame with a triangle_count column. Call dictionary_name['label'] to get the handle to frame whose vertex type is label.

28.82 *Commands* graph/ml/belief_propagation

Classification on sparse data using Belief Propagation.

28.82.1 POST /v1/commands/

28.82.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name graph/ml/belief_propagation

arguments **graph** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f66757730d0>>
<Missing Description>

prior_property : unicode

Name of the vertex property which contains the prior belief for the vertex.

posterior_property : unicode

Name of the vertex property which will contain the posterior belief for each vertex.

edge_weight_property : unicode (default=None)

Name of the edge property that contains the edge weight for each edge.

convergence_threshold : float64 (default=None)

Belief propagation will terminate when the average change in posterior beliefs between supersteps is less than or equal to this threshold.

max_iterations : int32 (default=None)

The maximum number of supersteps that the algorithm will execute. The valid range is all positive int.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Belief propagation by the sum-product algorithm. This algorithm analyzes a graphical model with prior beliefs using sum product message passing. The priors are read from a property in the graph, the posteriors are written to another property in the graph. This is the GraphX-based implementation of belief propagation.

See [Loopy Belief Propagation](#) for a more in-depth discussion of BP and LBP.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.82.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
dict
```

Progress report for belief propagation in the format of a multiple-line string.

28.83 *Commands* graph/rename

Rename a graph in the database.

28.83.1 POST /v1/commands/

28.83.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name graph/rename

arguments graph : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f66757730d0>>
<Missing Description>

new_name : unicode
the new name for the graph

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.83.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
dict
```

28.84 *Commands* model:collaborative_filtering/new

Collaborative filtering recommend model.

28.84.1 POST /v1/commands/

28.84.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:collaborative_filtering/new

arguments **dummy_model_ref** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

<Missing Description>

name : unicode (default=None)

User supplied name.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Collaborative Filtering

Collaborative filtering is a technique that is widely used in recommendation systems to suggest items (for example, products, movies, articles) to potential users based on historical records of items that users have purchased, rated, or viewed. The Trusted Analytics provides implementations of collaborative filtering with either Alternating Least Squares (ALS) or Conjugate Gradient Descent (CGD) optimization methods.

Both methods optimize the cost function found in Y. Koren, [Factorization Meets the Neighborhood: a Multifaceted Collaborative Filtering Model](#)⁸ in ACM KDD 2008. For more information on optimizing using ALS see, Y. Zhou, D. Wilkinson, R. Schreiber and R. Pan, [Large-Scale Parallel Collaborative Filtering for the Netflix Prize](#)⁹, 2008.

CGD provides a faster, more approximate optimization of the cost function and should be used when memory is a constraint.

A typical representation of the preference matrix P in Giraph is a bi-partite graph, where nodes at the left side represent a list of users and nodes at the right side represent a set of items (for example, movies), and edges encode the rating a user provided to an item. To support training, validation and test, a common practice in machine learning, each edge is also annotated by “TR”, “VA” or “TE”.

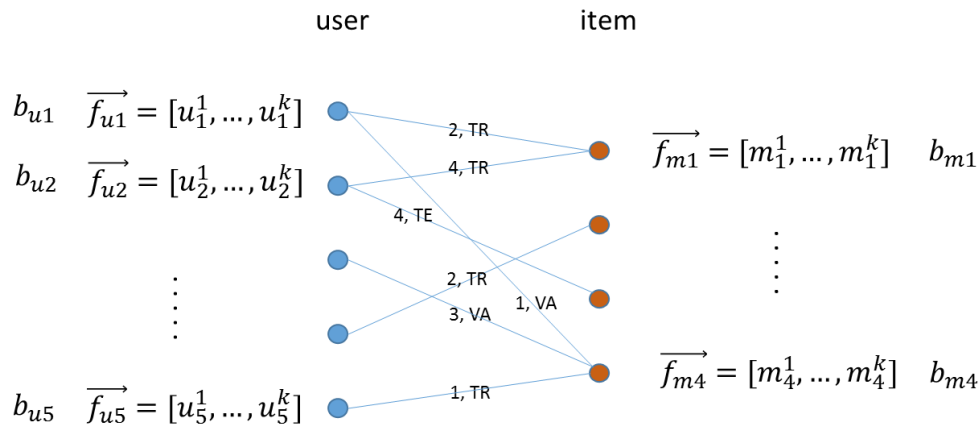


Fig. 28.1: A typical representation of the preference matrix P

Each node in the graph will be associated with a vector \vec{f}_x of length k , where k is the feature dimension specified by the user, and a bias term b_x . The predictions for item m_j , from user u_i are given by dot product of the feature vector and the user vector, plus the item and user bias terms: /home/work/atk/engine-plugins/giraph-plugins/src/main/scala/org/trustedanalytics/atk/giraph/plugins/model/cf/CollaborativeFilteringNewPlugin.scala

$$r_{ij} = \vec{f}_{u_i} \cdot \vec{f}_{m_j} + b_{u_i} + b_{m_j}$$

⁸<http://public.research.att.com/~volinsky/netflix/kdd08koren.pdf>

⁹<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.173.2797>

The parameters of the above equation are chosen to minimize the regularized mean squared error between known and predicted ratings:

$$cost = \frac{\sum error^2}{n} + \lambda * \left(bias^2 + \sum f_k^2 \right)$$

How this optimization is accomplished depends on whether the use uses the ALS or CGD functions respectively. It is recommended that the ALS method be used to solve collaborative filtering problems. The CGD method uses less memory than ALS, but it returns an approximate solution to the objective function and should only be used in cases when memory required for ALS is prohibitively high.

Using ALS Optimization to Solve the Collaborative Filtering Problem

ALS optimizes the vector \vec{f}_* and the bias b_* alternatively between user profiles using least squares on users and items. On the first iteration, the first feature of each item is set to its average rating, while the others are set to small random numbers. The algorithm then treats the m 's as constant and optimizes u_i^1, \dots, u_i^k for each user, i . For an individual user, this is a simple ordinary least squares optimization over the items that user has ranked. Next, the algorithm takes the u 's as constant and optimizes the m_j^1, \dots, m_j^k for each item, j . This is again an ordinary least squares optimization predicting the user rating of person that has ranked item j .

At each step, the bias is computed for either items or users and the objective function, shown below, is evaluated. The bias term for an item or user, computed for use in the next iteration is given by:

$$b = \frac{\sum error}{(1 + \lambda) * n}$$

The optimization is said to converge if the change in the objective function is less than the `convergence_threshold` parameter or the algorithm hits the maximum number of *supersteps*.

$$cost = \frac{\sum error^2}{n} + \lambda * \left(bias^2 + \sum f_k^2 \right)$$

Note that the equations above omit user and item subscripts for generality. The l_2 regularization term, lambda, tries to avoid overfitting by penalizing the magnitudes of the parameters, and λ is a tradeoff parameter that balances the two terms and is usually determined by cross validation (CV).

After the parameters \vec{f}_* and b_* are determined, given an item m_j the rating from user u_i can be predicted by the simple linear model:

$$r_{ij} = \vec{f}_{ui} \cdot \vec{f}_{mj} + b_{ui} + b_{mj}$$

Matrix Factorization based on Conjugate Gradient Descent (CGD)

This is the Conjugate Gradient Descent (CGD) with Bias for collaborative filtering algorithm. Our implementation is based on the paper:

Y. Koren. Factorization Meets the Neighborhood: a Multifaceted Collaborative Filtering Model. In ACM KDD 2008. (Equation 5) <http://public.research.att.com/~volinsky/netflix/kdd08koren.pdf>

This algorithm for collaborative filtering is used in *recommendation systems* to suggest items (products, movies, articles, and so on) to potential users based on historical records of items that all users have purchased, rated, or viewed. The records are usually organized as a preference matrix P , which is a sparse matrix holding the preferences (such as, ratings) given by users to items. Similar to ALS, CGD falls in the category of matrix factorization/latent factor model that infers user profiles and item profiles in low-dimension space, such that the original matrix P can be approximated by a linear model.

This factorization method uses the conjugate gradient method for its optimization subroutine. For more on conjugate gradient descent in general, see: http://en.wikipedia.org/wiki/Conjugate_gradient_method.

The Mathematics of Matrix Factorization via CGD

Matrix factorization by conjugate gradient descent produces ratings by using the (limited) space of observed rankings to infer a user-factors vector p_u for each user u , and an item-factors vector q_i for each item i , and then producing a ranking by user u of item i by the dot-product $b_{ui} + p_u^T q_i$ where b_{ui} is a baseline ranking calculated as $b_{ui} = \mu + b_u + b_i$.

The optimum model is chosen to minimum the following sum, which penalizes square distance of the prediction from observed rankings and complexity of the model (through the regularization term):

$$\sum_{(u,i) \in \mathcal{K}} (r_{ui} - \mu - b_u - b_i - p_u^T q_i)^2 + \lambda_3 (\|p_u\|^2 + \|q_i\|^2 + b_u^2 + b_i^2)$$

Where:

- r_{ui} — Observed ranking of item i by user u
- \mathcal{K} — Set of pairs (u, i) for each observed ranking of item i by user u
- μ — The average rating over all ratings of all items by all users.
- b_u — How much user u 's average rating differs from μ .
- b_i — How much item i 's average rating differs from μ
- p_u — User-factors vector.
- q_i — Item-factors vector.
- λ_3 — A regularization parameter specified by the user.

This optimization problem is solved by the conjugate gradient descent method. Indeed, this difference in how the optimization problem is solved is the primary difference between matrix factorization by CGD and matrix factorization by ALS.

Comparison between CGD and ALS

Both CGD and ALS provide recommendation systems based on matrix factorization; the difference is that CGD employs the conjugate gradient descent instead of least squares for its optimization phase. In particular, they share the same bipartite graph representation and the same cost function.

- ALS finds a better solution faster - when it can run on the cluster it is given.
- CGD has slighter memory requirements and can run on datasets that can overwhelm the ALS-based solution.

When feasible, ALS is a preferred solver over CGD, while CGD is recommended only when the application requires so much memory that it might be beyond the capacity of the system. CGD has a smaller memory requirement, but has a slower rate of convergence and can provide a rougher estimate of the solution than the more computationally intensive ALS.

The reason for this is that ALS solves the optimization problem by a least squares that requires inverting a matrix. Therefore, it requires more memory and computational effort. But ALS, a 2nd-order optimization method, enjoys higher convergence rate and is potentially more accurate in parameter estimation.

On the otherhand, CGD is a 1.5th-order optimization method that approximates the Hessian of the cost function from the previous gradient information through N consecutive CGD updates. This is very important in cases where the solution has thousands or even millions of components.

Usage

The matrix factorization by CGD procedure takes a property graph, encoding a bipartite user-item ranking network, selects a subset of the edges to be considered (via a selection of edge labels), takes initial ratings from specified edge property values, and then writes each user-factors vector to its user vertex in a specified vertex property name and each item-factors vector to its item vertex in the specified vertex property name.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.84.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of  
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
```

28.85 *Commands* model:collaborative_filtering/recommend

[BETA] Collaborative filtering (als/cgd) model

28.85.1 POST /v1/commands/

28.85.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:collaborative_filtering/recommend

arguments model : <bound method AtkEntityType.__name__ of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
 <Missing Description>

name : unicode

An entity name from the first column of the input frame

top_k : int32

positive integer representing the top recommendations for the name

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

see collaborative filtering train for more information

Response**Status**

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.85.3 GET /v1/commands/:id**Request****Route**

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
    see collaborative filtering train for more information
```

28.86 *Commands* model:collaborative_filtering/train

Collaborative filtering (als/cgd) model

28.86.1 POST /v1/commands/

28.86.2 GET /v1/commands:id

Request

Route

```
POST /v1/commands/
```

Body

name model:collaborative_filtering/train

arguments **model** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
<Missing Description>

user_col_name : unicode

Name of the user column from input data

item_col_name : unicode

Name of the item column from input data

rating_col_name : unicode

Name of the rating column from input data

evaluation_function : unicode (default=None)

als/cgd

num_factors : int32 (default=None)

Size of the desired factors (default is 3)

max_iterations : int32 (default=None)

Max number of iterations for Giraph

convergence_threshold : float64 (default=None)

float value between 0 .. 1

regularization : float32 (default=None)

float value between 0 .. 1

bias_on : bool (default=None)

bias on/off switch

min_value : float32 (default=None)

minimum edge weight value

max_value : float32 (default=None)

minimum edge weight value

learning_curve_interval : int32 (default=None)

iteration interval to output learning curve

cgd_iterations : int32 (default=None)

custom argument for cgd learning curve output interval (default: every iteration)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.86.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
dict
```

Execution result summary for Giraph

28.87 *Commands* model:k_means/new

create a new model

28.87.1 POST /v1/commands/

28.87.2 GET /v1/commands/:id

Request

Route


```
POST /v1/commands/
```

Body

name model:k_means/new

arguments **dummy_model_ref** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

name : unicode (default=None)
name for the model

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description**Response****Status**

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.87.3 GET /v1/commands/:id**Request****Route**

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
```

28.88 *Commands* model:k_means/predict

[BETA] Predict the cluster assignments for the data points.

28.88.1 POST /v1/commands/

28.88.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:k_means/predict

arguments model : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame whose labels are to be predicted. By default, predict is run on the same columns over which the model is trained.

observation_columns : list (default=None)

Column(s) containing the observations whose clusters are to be predicted. By default, we predict the clusters over columns the KMeansModel was trained on. The columns are scaled using the same values used when training the model.

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.88.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of  
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
```

Frame A new frame consisting of the existing columns of the frame and new columns. The data returned is composed of multiple components:

'k' columns [double] Containing squared distance of each point to every cluster center.

predicted_cluster [int] Integer containing the cluster assignment.

28.89 *Commands* model:k_means/publish

[BETA] Creates a tar file that will used as input to the scoring engine

28.89.1 POST /v1/commands/

28.89.2 GET /v1/commands/:id

Request

Route

POST /v1/commands/

Body

name model:k_means/publish

arguments model : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

Headers

Authorization: test_api_key_1 Content-type: application/json

Description

Returns the HDFS path to the tar file

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.89.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
dict
```

28.90 *Commands* model:k_means/train

[BETA] Creates KMeans Model from train frame.

28.90.1 POST /v1/commands/

28.90.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:k_means/train

arguments **model** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
A frame to train the model on.

observation_columns : list
Columns containing the observations.

column_scalings : list
Column scalings for each of the observation columns. The scaling value is multiplied by the corresponding value in the observation column.

k : int32 (default=None)
Desired number of clusters. Default is 2.

max_iterations : int32 (default=None)
Number of iterations for which the algorithm should run. Default is 20.

epsilon : float64 (default=None)
Distance threshold within which we consider k-means to have converged. Default is 1e-4.

initialization_mode : unicode (default=None)
The initialization technique for the algorithm. It could be either “random” or “k-meansll”. Default is “k-meansll”.

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Upon training the ‘k’ cluster centers are computed.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.90.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

dict

dict Results. The data returned is composed of multiple components:

cluster_size [dict] Cluster size

ClusterId [int] Number of elements in the cluster 'ClusterId'.

within_set_sum_of_squared_error [double] The set of sum of squared error for the model.

28.91 *Commands* model:lda/new

Creates Latent Dirichlet Allocation model

28.91.1 POST /v1/commands/

28.91.2 GET /v1/commands/:id

Request

Route

POST /v1/commands/

Body

name model:lda/new

arguments dummy_model_ref : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

name : unicode (default=None)
User supplied name.

Headers

Authorization: test_api_key_1 Content-type: application/json

Description

Topic Modeling with Latent Dirichlet Allocation

Topic modeling algorithms are a class of statistical approaches to partitioning items in a data set into subgroups. As the name implies, these algorithms are often used on corpora of textual data, where they are used to group documents in the collection into semantically-meaningful groupings. For an overall introduction to topic modeling, the reader might refer to the work of David Blei and Michael Jordan, who are credited with creating and popularizing topic modeling in the machine learning community. In particular, Blei's 2011 paper provides a nice introduction, and is freely-available online ¹⁰.

LDA is a commonly-used algorithm for topic modeling, but, more broadly, is considered a dimensionality reduction technique. It contrasts with other approaches (for example, latent semantic indexing), in that it creates what's referred to as a generative probabilistic model — a statistical model that allows the algorithm to generalize its approach to topic assignment to other, never-before-seen data points. For the purposes of exposition, we'll limit the scope of our discussion of LDA to the world of natural language processing, as it has an intuitive use there (though LDA can be used on other types of data). In general, LDA represents documents as random mixtures over topics in the corpus. This makes sense because any work of writing is rarely about a single subject. Take the case of a news article on the President of the United States of America's approach to healthcare as an example. It would be reasonable to assign topics like President, USA, health insurance, politics, or healthcare to such a work, though it is likely to primarily discuss the President and healthcare.

¹⁰ <http://www.cs.princeton.edu/~blei/papers/Blei2011.pdf>

LDA assumes that input corpora contain documents pertaining to a given number of topics, each of which are associated with a variety of words, and that each document is the result of a mixture of probabilistic samplings: first over the distribution of possible topics for the corpora, and second over the list of possible words in the selected topic. This generative assumption confers one of the main advantages LDA holds over other topic modeling approaches, such as probabilistic and regular LSI. As a generative model, LDA is able to generalize the model it uses to separate documents into topics to documents outside the corpora. For example, this means that using LDA to group online news articles into categories like Sports, Entertainment, and Politics, it would be possible to use the fitted model to help categorize newly-published news stories. Such an application is beyond the scope of approaches like LSI. What's more, when fitting an LSI model, the number of parameters that have to be estimated scale linearly with the number of documents in the corpus, whereas the number of parameters to estimate for an LDA model scales with the number of topics — a much lower number, making it much better-suited to working with large data sets.

The Typical Latent Dirichlet Allocation Workflow

Although every user is likely to have his or her own habits and preferred approach to topic modeling a document corpus, there is a general workflow that is a good starting point when working with new data. The general steps to the topic modeling with LDA include:

1. Data preparation and ingest
2. Assignment to training or testing partition
3. Graph construction
4. Training LDA
5. Evaluation
6. Interpretation of results

Data preparation and ingest

Most topic modeling workflows involve several data pre-processing and cleaning steps. Depending on the characteristics of the data being analyzed, there are different best-practices to use here, so it's important to be familiar with the standard procedures for analytics in the domain from which the text originated. For example, in the biomedical text analytics community, it is common practice for text analytics workflows to involve pre-processing for identifying negation statements (Chapman et al., 2001¹¹). The reason for this is many analysts in that domain are examining text for diagnostic statements — thus, failing to identify a negated statement in which a disease is mentioned could lead to undesirable false-positives, but this phenomenon may not arise in every domain. In general, both stemming and stop word filtering are recommended steps for topic modeling pre-processing. Stemming refers to a set of methods used to normalize different tenses and variations of the same word (for example, stemmer, stemming, stemmed, and stem). Stemming algorithms will normalize all variations of a word to one common form (for example, stem). There are many approaches to stemming, but the Porter Stemming (Porter, 2006¹²) is one of the most commonly-used.

Removing common, uninformative words, or stop word filtering, is another commonly-used step in data pre-processing for topic modeling. Stop words include words like *the*, *and*, or *a*, but the full list of uninformative words can be quite long and depend on the domain producing the text in question. Example stop word lists online¹³ can be a great place to start, but being aware of the best-practices in the applicable field is necessary to expand upon these.

There may be other pre-processing steps needed, depending on the type of text being worked with. Punctuation removal is frequently recommended, for example. To determine what's best for the text being analyzed, it helps to understand a bit about how LDA analyzes the input text. To learn the topic model, LDA will typically look at the frequency of individual words across documents, which are determined based on space-separation. Thus, each word will be interpreted independent of where it occurs in a document, and without regard for the words that were written around it. In the text analytics field, this is often referred to as a *bag of words* approach to tokenization, the

¹¹ <http://www.sciencedirect.com/science/article/pii/S1532046401910299>

¹² <http://tartarus.org/~martin/PorterStemmer/index.html>

¹³ <http://www.textfixer.com/resources/common-english-words.txt>

process of separating input text into composite features to be analyzed by some algorithm. When choosing pre-processing steps, it helps to keep this in mind. Don't worry too much about removing words or modifying their format — you're not manipulating your data! These steps simply make it easier for the topic modeling algorithm to find the latent topics that comprise your corpus.

Assignment to training or testing partition

The random assignment to training and testing partitions is an important step in most every machine learning workflow. It is common practice to withhold a random selection of one's data set for the purpose of evaluating the accuracy of the model that was learned from the training data. The results of this evaluation allow the user to confidently speak about the generalizability of the trained model. When speaking in these terms, be cautious that you only discuss generalizability to the broader population from which your data was originally obtained. If a topic model is trained on neuroscience-related publications, for example, evaluating the model on other neuroscience-related publications is valid. It would not be valid to discuss the model's ability to work on documents from other domains.

There are various schools of thought for how to assign a data set to training and testing collections, but all agree that the process should be random. Where analysts disagree is in the ratio of data to be assigned to each. In most situations, the bulk of data will be assigned to the training collection, because the more data that can be used to train the algorithm, the better the resultant model will typically be. It's also important that the testing collection have sufficient data to be able to reflect the characteristics of the larger population from which it was drawn (this becomes an important issue when working with data sets with rare topics, for example). As a starting point, many people will use a 90%/10% training/test collection split, and modify this ratio based on the characteristics of the documents being analyzed.

Graph construction

Trusted Analytics uses a bipartite graph, to learn an LDA topic model. This graph contains vertices in two columns. The left-hand column contains unique ids, each corresponding to a document in the training collection, while the right-hand column contains unique ids corresponding to each word in the entire training set, following any pre-processing steps that were used. Connections between these columns, or edges, denote the number of times a particular word appears in a document, with the weight on the edge in question denoting the number of times the word was found there. After graph construction, many analysts choose to normalize the weights using one of a variety of normalization schemes. One approach is to normalize the weights to sum to 1, while another is to use an approach called term frequency-inverse document frequency (tfidf), where the resultant weights are meant to reflect how important a word is to a document in the corpus. Whether to use normalization — or what technique to use — is an open question, and will likely depend on the characteristics of the text being analyzed. Typical text analytics experiments will try a variety of approaches on a small subset of the data to determine what works best.

See [Figure 1](#).

Training the Model

In using LDA, we are trying to model a document collection in terms of topics $\beta_{1:K}$, where each β_K describes a distribution over the set of words in the training corpus. Every document d , then, is a vector of proportions θ_d , where $\theta_{d,k}$ is the proportion of the d^{th} document for topic k . The topic assignment for document d is z_d , and $z_{d,n}$ is the topic assignment for the n^{th} word in document d . The words observed in document d are $w_{d,n}$, and $w_{d,n}$ is the n^{th} word in document d . The generative process for LDA, then, is the joint distribution of hidden and observed values

$$p(\beta_{1:K}, \theta_{1:D}, z_{1:D}, w_{1:D}) = \prod_{i=1}^K p(\beta_i) \prod_{i=1}^D p(\theta_i) \left(\prod_{n=1}^N p(z_{d,n} | \theta_d) p(w_{d,n} | \beta_{1:K}, z_{d,n}) \right)$$

This distribution depicts several dependencies: topic assignment $z_{d,n}$ depends on the topic proportions θ_d , and the observed word $w_{d,n}$ depends on topic assignment $z_{d,n}$ and all the topics $\beta_{1:K}$, for example. Although there are no analytical solutions to learning the LDA model, there are a variety of approximate solutions that are used, most of which are based on Gibbs Sampling (for example, Porteous et al., 2008¹⁴). The Trusted Analytics uses an

¹⁴ <http://www.ics.uci.edu/~newman/pubs/fastlda.pdf>

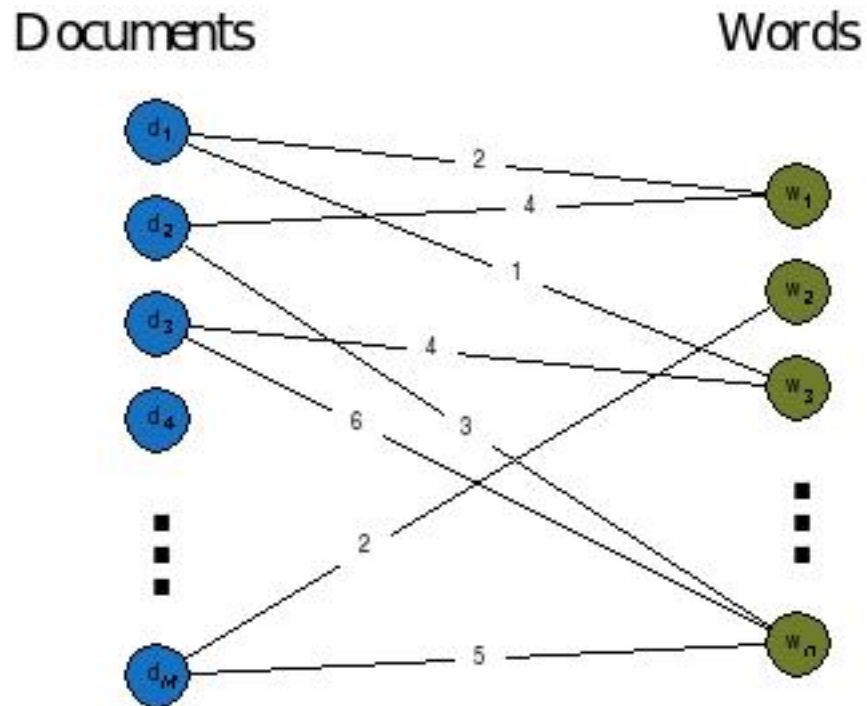


Fig. 28.2: Figure 1 - Example layout of a bipartite graph for LDA.

The left-hand column contains one vertex for each document in the input corpus, while the right-hand column contains vertices for each unique word found in them. Edges connecting left- and right-hand columns denote the number of times the word was found in the document the edge connects. The weights of the edges used in this example were not normalized.

implementation related to this. We refer the interested reader to the primary source on this approach to learn more (Teh et al., 2006¹⁵).

Evaluation

As with every machine learning algorithm, evaluating the accuracy of the model that has been obtained is an important step before interpreting the results. With many types of algorithms, the best practices in this step are straightforward — in supervised classification, for example, we know the true labels of the data being classified, so evaluating performance can be as simple as computing the number of errors, calculating receiver operating characteristic, or F1 measure. With topic modeling, the situation is not so straightforward. This makes sense, if we consider with LDA we're using an algorithm to blindly identify logical subgroupings in our data, and we don't *a priori* know the best grouping that can be found. Evaluation, then, should proceed with this in mind, and an examination of homogeneity of the words comprising the documents in each grouping is often done. This issue is discussed further in Blei's 2011 introduction to topic modeling¹⁶. It is of course possible to evaluate a topic model from a statistical perspective using our hold-out testing document collection — and this is a recommended best practice — however, such an evaluation does not assess the topic model in terms of how they are typically used.

Interpretation of results

After running LDA on a document corpus, users will typically examine the top n most frequent words that can be found in each grouping. With this information, one is often able to use their own domain expertise to think of logical names for each topic (this situation is analogous to the step in principal components analysis, wherein statisticians will think of logical names for each principal component based on the mixture of dimensions each spans). Each document, then, can be assigned to a topic, based on the mixture of topics it has been assigned. Recall that LDA will assign each document a set of probabilities corresponding to each possible topic. Researchers will often set some threshold value to make a categorical judgment regarding topic membership, using this information.

footnotes

Response

Status

200 OK

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.91.3 GET /v1/commands/:id

Request

Route

GET /v1/commands/18

Body

(None)

¹⁵ http://machinelearning.wustl.edu/mlpapers/paper_files/NIPS2006_511.pdf

¹⁶ <http://www.cs.princeton.edu/~blei/papers/Blei2011.pdf>

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response**Status**

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
```

28.92 *Commands* model:lda/predict

[BETA] Predict conditional probabilities of topics given document.

28.92.1 POST /v1/commands/

28.92.2 GET /v1/commands/:id

Request**Route**

```
POST /v1/commands/
```

Body

name model:lda/predict

arguments model : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

Reference to the model for which topics are to be determined.

document : list

Document whose topics are to be predicted.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Predicts conditional probabilities of topics given document using trained Latent Dirichlet Allocation model. The input document is represented as a list of strings

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.92.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

dict

dict Dictionary containing predicted topics. The data returned is composed of multiple components:

topics_given_doc [list of doubles] List of conditional probabilities of topics given document.

new_words_count [int] Count of new words in test document not present in training set.

new_words_percentage: double Percentage of new words in test document.

28.93 *Commands* model:lda/publish

[BETA] Creates a tar file that will used as input to the scoring engine

28.93.1 POST /v1/commands/

28.93.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:lda/publish

arguments model : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Creates a tar file with the trained Latent Dirichlet Allocation model. The tar file is used as input to the scoring engine to predict the conditional topic probabilities for a document.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.93.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

dict

Returns the HDFS path to the tar file

28.94 *Commands* model:lda/train

[BETA] Creates Latent Dirichlet Allocation model

28.94.1 POST /v1/commands/

28.94.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:lda/train

arguments **model** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

Input frame data.

document_column_name : unicode

Column Name for documents. Column should contain a str value.

word_column_name : unicode

Column name for words. Column should contain a str value.

word_count_column_name : unicode

Column name for word count. Column should contain an int32 or int64 value.

max_iterations : int32 (default=None)

The maximum number of iterations that the algorithm will execute. The valid value range is all positive int. Default is 20.

alpha : float32 (default=None)

The hyper-parameter for document-specific distribution over topics. Mainly used as a smoothing parameter in *Bayesian inference*. Larger value implies that documents are assumed to cover all topics more uniformly; smaller value implies that documents are more concentrated on a small subset of topics. Valid value range is all positive float.

Default is 0.1.

beta : float32 (default=None)

The hyper-parameter for word-specific distribution over topics. Mainly used as a smoothing parameter in *Bayesian inference*. Larger value implies that topics contain all words more uniformly and smaller value implies that topics are more concentrated on a small subset of words. Valid value range is all positive float. Default is 0.1.

convergence_threshold : float32 (default=None)

The amount of change in LDA model parameters that will be tolerated at convergence. If the change is less than this threshold, the algorithm exits before it reaches the maximum number of supersteps. Valid value range is all positive float and 0.0. Default is 0.001.

evaluate_cost : bool (default=None)

“True” means turn on cost evaluation and “False” means turn off cost evaluation. It’s relatively expensive for LDA to evaluate cost function. For time-critical applications, this option allows user to turn off cost function evaluation. Default is “False”.

num_topics : int32 (default=None)

The number of topics to identify in the LDA model. Using fewer topics will speed up the computation, but the extracted topics might be more abstract or less specific; using more topics will result in more computation but lead to more specific topics. Valid value range is all positive int. Default is 10.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

See the discussion about [Latent Dirichlet Allocation at Wikipedia](http://en.wikipedia.org/wiki/Latent_Dirichlet_allocation).¹⁷

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.94.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
dict
```

¹⁷http://en.wikipedia.org/wiki/Latent_Dirichlet_allocation

dict The data returned is composed of multiple components:

topics_given_doc [Frame] Frame with conditional probabilities of topic given document.

word_given_topics [Frame] Frame with conditional probabilities of word given topic.

topics_given_word [Frame] Frame with conditional probabilities of topic given word.

report [str] The configuration and learning curve report for Latent Dirichlet Allocation as a multiple line str.

28.95 *Commands* model:libsvm/new

[ALPHA] model:libsvm/new

28.95.1 POST /v1/commands/

28.95.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:libsvm/new

arguments **dummy_model_ref** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

name : unicode (default=None)

User supplied name.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.95.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of  
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
```

28.96 *Commands* model:libsvm/predict

[ALPHA] New frame with new predicted label column.

28.96.1 POST /v1/commands/

28.96.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:libsvm/predict

arguments **model** : <bound method AtkEntityType.__name__ of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
 <Missing Description>

frame : <bound method AtkEntityType.__name__ of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
 A frame whose labels are to be predicted.

observation_columns : list (default=None)
 Column(s) containing the observations whose labels are to be predicted. Default is the
 columns the LibsvmModel was trained on.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Predict the labels for a test frame and create a new frame revision with existing columns and a new predicted label's column.

Response**Status**

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.96.3 GET /v1/commands/:id**Request****Route**

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
```

A new frame containing the original frame's columns and a column *predicted_label* containing the score calculated for each observation.

28.97 *Commands* model:libsvm/publish

[BETA] Creates a tar file that will used as input to the scoring engine

28.97.1 POST /v1/commands/

28.97.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:libsvm/publish

arguments model : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Returns the HDFS path to the tar file

Response**Status**

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.97.3 GET /v1/commands/:id**Request****Route**

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response**Status**

```
200 OK
```

Body

```
dict
```

28.98 *Commands* model:libsvm/score

[ALPHA] Calculate the prediction label for a single observation.

28.98.1 POST /v1/commands/

28.98.2 GET /v1/commands/:id

Request

Route

POST /v1/commands/

Body

name model:libsvm/score

arguments **model** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

vector : None
<Missing Description>

Headers

Authorization: test_api_key_1
Content-type: application/json

Description

Response

Status

200 OK

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.98.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

dict

Predicted label.

28.99 *Commands* model:libsvm/test

[ALPHA] Predict test frame labels and return metrics.

28.99.1 POST /v1/commands/

28.99.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:libsvm/test

arguments **model** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame whose labels are to be predicted.

label_column : unicode

Column containing the actual label for each observation.

observation_columns : list (default=None)

Column(s) containing the observations whose labels are to be predicted and tested.
Default is to test over the columns the LibsvmModel was trained on.

Headers

Authorization: test_api_key_1 Content-type: application/json

Description

Predict the labels for a test frame and run classification metrics on predicted and target labels.

Response

Status

200 OK

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.99.3 GET /v1/commands/:id

Request

Route

GET /v1/commands/18

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
dict
```

Object Object with binary classification metrics. The data returned is composed of multiple components:

<object>.accuracy [double] The degree of correctness of the test frame labels.

<object>.confusion_matrix [table] A specific table layout that allows visualization of the performance of the test.

<object>.f_measure [double] A measure of a test's accuracy. It considers both the precision and the recall of the test to compute the score.

<object>.precision [double] The degree to which the correctness of the label is expressed.

<object>.recall [double] The fraction of relevant instances that are retrieved.

28.100 *Commands* model:libsvm/train

[ALPHA] Train Lib Svm model based on another frame.

28.100.1 POST /v1/commands/

28.100.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:libsvm/train

arguments **model** : <bound method AtkEntityType.__name__ of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
 <Missing Description>

frame : <bound method AtkEntityType.__name__ of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame to train the model on.

label_column : unicode

Column name containing the label for each observation.

observation_columns : list

Column(s) containing the observations.

svm_type : int32 (default=2)

Set type of SVM. Default is one-class SVM.

0 – C-SVC 1 – nu-SVC 2 – one-class SVM 3 – epsilon-SVR 4 – nu-SVR

kernel_type : int32 (default=2)

Specifies the kernel type to be used in the algorithm. Default is RBF.

0 – linear: $u \cdot v$ 1 – polynomial: $(\gamma u \cdot v + \text{coef0})^{\text{degree}}$ 2 – radial basis function: $\exp(-\gamma \|u - v\|^2)$ 3 – sigmoid: $\tanh(\gamma u \cdot v + \text{coef0})$

weight_label : list (default=None)

Default is (Array[Int](0))

weight : list (default=None)

Default is (Array[Double](0.0))

epsilon : float64 (default=0.001)

Set tolerance of termination criterion

degree : int32 (default=3)

Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

gamma : float64 (default=None)

Kernel coefficient for 'rbf', 'poly' and 'sigmoid'. Default is $1/n_{\text{features}}$.

coef : float64 (default=0.0)

Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

nu : float64 (default=0.5)

Set the parameter nu of nu-SVC, one-class SVM, and nu-SVR.

cache_size : float64 (default=100.0)

Specify the size of the kernel cache (in MB).

shrinking : int32 (default=1)

Whether to use the shrinking heuristic. Default is 1 (true).

probability : int32 (default=0)

Whether to enable probability estimates. Default is 0 (false).

nr_weight : int32 (default=1)

NR Weight

c : float64 (default=1.0)

Penalty parameter c of the error term.

p : float64 (default=0.1)

Set the epsilon in loss function of epsilon-SVR.

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Creating a lib Svm Model using the observation column and label column of the train frame.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.100.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
_Unit
```

28.101 *Commands* model:linear_regression/new

28.101.1 POST /v1/commands/

28.101.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:linear_regression/new

arguments **dummy_model_ref** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

<Missing Description>

name : unicode (default=None)

User supplied name.

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.101.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of  
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
```

28.102 *Commands* model:linear_regression/predict

[ALPHA] Make new frame with column for label prediction.

28.102.1 POST /v1/commands/

28.102.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:linear_regression/predict

arguments **model** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame whose labels are to be predicted. By default, predict is run on the same columns over which the model is trained.

observation_columns : list (default=None)

Column(s) containing the observations whose labels are to be predicted. By default, we predict the labels over columns the LogisticRegressionModel was trained on.

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Predict the labels for a test frame and create a new frame revision with existing columns and a new predicted label's column.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.102.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
```

Frame containing the original frame's columns and a column with the predicted label.

28.103 *Commands* model:linear_regression/train

[ALPHA] Build linear regression model.

28.103.1 POST /v1/commands/

28.103.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:linear_regression/train

arguments **model** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

<Missing Description>

frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame to train the model on.

label_column : unicode

Column name containing the label for each observation.

observation_columns : list

Column(s) containing the observations.

intercept : bool (default=None)

The algorithm adds an intercept. Default is true.

num_iterations : int32 (default=None)

Number of iterations. Default is 100.

step_size : int32 (default=None)

Step size for optimizer. Default is 1.0.

reg_type : unicode (default=None)

Regularization L1 or L2. Default is L2.

reg_param : float64 (default=None)

Regularization parameter. Default is 0.01.

mini_batch_fraction : float64 (default=None)

Mini batch fraction parameter. Default is 1.0.

Headers

Authorization: test_api_key_1 Content-type: application/json

Description

Creating a LinearRegression Model using the observation column and label column of the train frame.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.103.3 GET /v1/commands/:id

Request**Route**

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response**Status**

```
200 OK
```

Body

```
_Unit
```

28.104 *Commands* model:logistic_regression/new

Create a ‘new’ instance of logistic regression model.

28.104.1 POST /v1/commands/

28.104.2 GET /v1/commands/:id

Request**Route**

```
POST /v1/commands/
```

Body

```
name model:logistic_regression/new
```

arguments **dummy_model_ref** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>
name : unicode (default=None)
User supplied name.

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.104.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
```

28.105 *Commands* model:logistic_regression/predict

[ALPHA] Make a new frame with a column for label prediction.

28.105.1 POST /v1/commands/

28.105.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:logistic_regression/predict

arguments **model** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame whose labels are to be predicted. By default, predict is run on the same columns over which the model is trained.

observation_columns : list (default=None)

Column(s) containing the observations whose labels are to be predicted. By default, we predict the labels over columns the LogisticRegressionModel was trained on.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Predict the labels for a test frame and create a new frame revision with existing columns and a new predicted label's column.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.105.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
```

Frame containing the original frame's columns and a column with the predicted label.

28.106 *Commands* model:logistic_regression/test

[ALPHA] Predict test frame labels and show metrics.

28.106.1 POST /v1/commands/

28.106.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:logistic_regression/test

arguments model : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
frame whose labels are to be predicted.

label_column : unicode

Column containing the actual label for each observation.

observation_columns : list (default=None)

Column(s) containing the observations whose labels are to be predicted and tested.
Default is to test over the columns the SvmModel was trained on.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Predict the labels for a test frame and run classification metrics on predicted and target labels.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.106.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

dict

object An object with binary classification metrics. The data returned is composed of multiple components:

```
<object>.accuracy : double <object>.confusion_matrix : table <object>.f_measure : double
<object>.precision : double <object>.recall : double
```

28.107 *Commands* model:logistic_regression/train

[ALPHA] Build logistic regression model.

28.107.1 POST /v1/commands/

28.107.2 GET /v1/commands/:id

Request

Route

POST /v1/commands/

Body**name** model:logistic_regression/train

arguments model : <bound method AtkEntityType.__name__ of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
 <Missing Description>

frame : <bound method AtkEntityType.__name__ of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
 A frame to train the model on.

label_column : unicode
 Column name containing the label for each observation.

observation_columns : list
 Column(s) containing the observations.

frequency_column : unicode (default=None)
 Optional column containing the frequency of observations.

num_classes : int32 (default=2)
 Number of classes

optimizer : unicode (default=LBFGS)
Set type of optimizer. LBFGS - Limited-memory BFGS. LBFGS supports multinomial logistic regression. SGD - Stochastic Gradient Descent. SGD only supports binary logistic regression.

compute_covariance : bool (default=True)
 If true, compute covariance matrix for the model.

intercept : bool (default=True)
 If true, add intercept column to training data.

feature_scaling : bool (default=False)
 If true, perform feature scaling before training model.

threshold : float64 (default=0.5)
 Threshold for separating positive predictions from negative predictions.

reg_type : unicode (default=L2)
Set type of regularization L1 - L1 regularization with sum of absolute values of coefficients L2 - L2 regularization with sum of squares of coefficients

reg_param : float64 (default=0.0)
 Regularization parameter

num_iterations : int32 (default=100)
 Maximum number of iterations

convergence_tolerance : float64 (default=0.0001)

Convergence tolerance of iterations for L-BFGS. Smaller value will lead to higher accuracy with the cost of more iterations.

num_corrections : int32 (default=10)

Number of corrections used in LBFGS update. Default 10. Values of numCorrections less than 3 are not recommended; large values of numCorrections will result in excessive computing time.

mini_batch_fraction : float64 (default=1.0)

Fraction of data to be used for each SGD iteration

step_size : int32 (default=1)

Initial step size for SGD. In subsequent steps, the step size decreases by $\text{stepSize}/\sqrt{t}$

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Creating a LogisticRegression Model using the observation column and label column of the train frame.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.107.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response**Status**

200 OK

Body

dict

object An object with a summary of the trained model. The data returned is composed of multiple components:

numFeatures [Int] Number of features in the training data

numClasses [Int] Number of classes in the training data

summaryTable: table A summary table composed of:

covarianceMatrix: Frame (optional) Covariance matrix of the trained model. The covariance matrix is the inverse of the Hessian matrix for the trained model. The Hessian matrix is the second-order partial derivatives of the model's log-likelihood function

“”

28.108 *Commands* model:naive_bayes/new

create a new model

28.108.1 POST /v1/commands/

28.108.2 GET /v1/commands/:id

Request**Route**

POST /v1/commands/

Body

name model:naive_bayes/new

arguments **dummy_model_ref**: <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

<Missing Description>

name : unicode (default=None)

User supplied name.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.108.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
```

28.109 *Commands* model:naive_bayes/predict

[ALPHA] Predict

28.109.1 POST /v1/commands/

28.109.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:naive_bayes/predict

arguments **model** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame whose labels are to be predicted. By default, predict is run on the same columns over which the model is trained.

observation_columns : list (default=None)

Column(s) containing the observations whose labels are to be predicted. By default, we predict the labels over columns the NaiveBayesModel was trained on.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.109.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of  
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
```

28.110 *Commands* model:naive_bayes/train

[ALPHA] Build a naive bayes model.

28.110.1 POST /v1/commands/

28.110.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:naive_bayes/train

arguments **model** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
A frame to train the model on.

label_column : unicode
Column containing the label for each observation.

observation_columns : list
Column(s) containing the observations.

lambda_parameter : float64 (default=None)
Additive smoothing parameter Default is 1.0.

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Train a NaiveBayesModel using the observation column, label column of the train frame and an optional lambda value.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.110.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
_Unit
```

28.111 *Commands* model:principal_components/new

Create a 'new' instance of principal component model.

28.111.1 POST /v1/commands/

28.111.2 GET /v1/commands/:id

Request

Route


```
POST /v1/commands/
```

Body

name model:principal_components/new

arguments **dummy_model_ref** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

name : unicode (default=None)

User supplied name.

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description**Response****Status**

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.111.3 GET /v1/commands/:id**Request****Route**

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
```

28.112 *Commands* model:principal_components/predict

[ALPHA] Predict using principal components model.

28.112.1 POST /v1/commands/

28.112.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:principal_components/predict

arguments model : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

Handle to the model to be used.

frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

Frame whose principal components are to be computed.

mean_centered : bool (default=True)

Option to mean center the columns. Default is true

t_squared_index : bool (default=False)

Indicator for whether the t-square index is to be computed. Default is false.

observation_columns : list (default=None)

List of observation column name(s) to be used for prediction. Default is the list of column name(s) used to train the model.

c : int32 (default=None)

The number of principal components to be predicted. Default is the count used to train the model.

name : unicode (default=None)

The name of the output frame generated by predict.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Predicting on a dataframe's columns using a PrincipalComponents Model.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.112.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

200 OK

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
```

A frame with existing columns and 'c' additional columns containing the projections of V on the the frame and an additional column storing the t-square-index value if requested

28.113 *Commands* model:principal_components/publish

[BETA] Creates a tar file that will be used as input to the scoring engine

28.113.1 POST /v1/commands/

28.113.2 GET /v1/commands/:id

Request

Route

POST /v1/commands/

Body

name model:principal_components/publish

arguments model : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

Headers

Authorization: test_api_key_1 Content-type: application/json

Description

Creates a tar file with the trained Principal Components Model. The tar file is used as input to the scoring engine to compute the principal components and t-squared index of the observation.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.113.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
dict
```

Returns the HDFS path to the tar file

28.114 *Commands* model:principal_components/train

Build principal components model.

28.114.1 POST /v1/commands/

28.114.2 GET /v1/commands/:id

Request

Route

POST /v1/commands/

Body

name model:principal_components/train

arguments model : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
Handle to the model to be used.

frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
A frame to train the model on.

observation_columns : list
List of column(s) containing the observations.

mean_centered : bool (default=True)
Option to mean center the columns

k : int32 (default=None)
Principal component count. Default is the number of observation columns

Headers

Authorization: test_api_key_1 Content-type: application/json

Description

Creating a PrincipalComponents Model using the observation columns.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.114.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
dict
```

28.115 *Commands* model:random_forest_classifier/new

Create a 'new' instance of random forest classifier model.

28.115.1 POST /v1/commands/

28.115.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:random_forest_classifier/new

arguments dummy_model_ref : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

name : unicode (default=None)

User supplied name.

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.115.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers


```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
```

28.116 *Commands* model:random_forest_classifier/predict

[ALPHA] Predict the labels for the data points.

28.116.1 POST /v1/commands/

28.116.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:random_forest_classifier/predict

arguments model : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

Handle of the model to be used

frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame whose labels are to be predicted. By default, predict is run on the same columns over which the model is trained.

observation_columns : list (default=None)

Column(s) containing the observations whose labels are to be predicted. By default, we predict the labels over columns the RandomForestModel was trained on.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.116.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
```

Frame A new frame consisting of the existing columns of the frame and a new column with predicted label for each observation.

28.117 *Commands* model:random_forest_classifier/publish

[BETA] Creates a tar file that will be used as input to the scoring engine

28.117.1 POST /v1/commands/

28.117.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:random_forest_classifier/publish

arguments model : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Creates a tar file with the trained Random Forest Classifier Model The tar file is used as input to the scoring engine to predict the class of an observation.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.117.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
dict
```

Returns the HDFS path to the tar file

28.118 *Commands* model:random_forest_classifier/test

[ALPHA] Predict test frame labels and return metrics.

28.118.1 POST /v1/commands/

28.118.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:random_forest_classifier/test

arguments model : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

Handle of the model to be used

frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

The frame whose labels are to be predicted

label_column : unicode

Column containing the true labels of the observations

observation_columns : list (default=None)

Column(s) containing the observations whose labels are to be predicted. By default, we predict the labels over columns the RandomForest was trained on.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Predict the labels for a test frame and run classification metrics on predicted and target labels.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.118.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response**Status**

```
200 OK
```

Body

dict

object

An object with classification metrics. The data returned is composed of multiple components:

```
<object>.accuracy : double <object>.confusion_matrix : table <object>.f_measure : double
<object>.precision : double <object>.recall : double
```

28.119 *Commands* model:random_forest_classifier/train

[ALPHA] Build Random Forests Classifier model.

28.119.1 POST /v1/commands/

28.119.2 GET /v1/commands/:id

Request**Route**

```
POST /v1/commands/
```

Body

name model:random_forest_classifier/train

arguments **model** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

Handle to the model to be used.

frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame to train the model on.

label_column : unicode

Column name containing the label for each observation.

observation_columns : list

Column(s) containing the observations.

num_classes : int32 (default=2)

Number of classes for classification

num_trees : int32 (default=1)

Number of trees in the random forest

impurity : unicode (default=gini)

Criterion used for information gain calculation. Supported values “gini” or “entropy”

max_depth : int32 (default=4)

Maximum depth of the tree

max_bins : int32 (default=100)

Maximum number of bins used for splitting features

seed : int32 (default=-842726101)

Random seed for bootstrapping and choosing feature subsets

categorical_features_info : None (default=None)

<Missing Description>

feature_subset_category : unicode (default=None)

Number of features to consider for splits at each node. Supported values “auto”, “all”, “sqrt”, “log2”, “onethird”

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Creating a Random Forests Classifier Model using the observation columns and label column.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.119.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
dict
```

28.120 *Commands* model:random_forest_regressor/new

<Missing Doc>

28.120.1 POST /v1/commands/

28.120.2 GET /v1/commands/:id

Request

Route


```
POST /v1/commands/
```

Body

name model:random_forest_regressor/new

arguments dummy_model_ref : <bound method AtkEntityType.__name__ of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
 <Missing Description>

name : unicode (default=None)

User supplied name.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description**Response****Status**

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.120.3 GET /v1/commands/:id**Request****Route**

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
```

28.121 *Commands* model:random_forest_regressor/predict

[ALPHA] Predict the values for the data points.

28.121.1 POST /v1/commands/

28.121.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:random_forest_regressor/predict

arguments model : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>

Handle of the model to be used

frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame whose labels are to be predicted. By default, predict is run on the same columns over which the model is trained.

observation_columns : list (default=None)

Column(s) containing the observations whose labels are to be predicted. By default, we predict the labels over columns the RandomForestModel was trained on.

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.121.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of  
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
```

Frame A new frame consisting of the existing columns of the frame and a new column with predicted value for each observation.

28.122 *Commands* model:random_forest_regressor/publish

[BETA] Creates a tar file that will be used as input to the scoring engine

28.122.1 POST /v1/commands/

28.122.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:random_forest_regressor/publish

arguments **model** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Creates a tar file with the trained Random Forest Regressor Model The tar file is used as input to the scoring engine to predict the value of an observation.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.122.3 GET /v1/commands/:id**Request****Route**

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response**Status**

```
200 OK
```

Body

```
dict
```

Returns the HDFS path to the tar file

28.123 *Commands* model:random_forest_regressor/train

[ALPHA] Build Random Forests Regressor model.

28.123.1 POST /v1/commands/**28.123.2 GET /v1/commands/:id****Request****Route**

```
POST /v1/commands/
```

Body

name model:random_forest_regressor/train

arguments **model** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
Handle to the model to be used.

frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
A frame to train the model on

label_column : unicode
Column name containing the label for each observation

observation_columns : list
Column(s) containing the observations

num_trees : int32 (default=1)
Number of trees in the random forest

impurity : unicode (default=variance)
Criterion used for information gain calculation. Supported values “variance”

max_depth : int32 (default=4)
Maximum depth of the tree

max_bins : int32 (default=100)
Maximum number of bins used for splitting features

seed : int32 (default=284863502)
Random seed for bootstrapping and choosing feature subsets

categorical_features_info : None (default=None)
<Missing Description>

feature_subset_category : unicode (default=None)
Number of features to consider for splits at each node. Supported values “auto”, “all”,
“sqrt”, “log2”, “onethird”

Headers

Authorization: test_api_key_1 Content-type: application/json

Description

Creating a Random Forests Regressor Model using the observation columns and label column.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.123.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
dict
```

28.124 *Commands* model:svm/new

[ALPHA] create a new model

28.124.1 POST /v1/commands/

28.124.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:svm/new

arguments dummy_model_ref : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

name : unicode (default=None)

User supplied name.

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.124.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers


```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
```

28.125 *Commands* model:svm/predict

[ALPHA] Make new frame with additional column for predicted label.

28.125.1 POST /v1/commands/

28.125.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:svm/predict

arguments model : <bound method AtkEntityType.__name__ of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
 <Missing Description>

frame : <bound method AtkEntityType.__name__ of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>

A frame whose labels are to be predicted. By default, predict is run on the same columns over which the model is trained.

observation_columns : list (default=None)

Column(s) containing the observations whose labels are to be predicted. By default, we predict the labels over columns the LogisticRegressionModel was trained on.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Predict the labels for a test frame and create a new frame revision with existing columns and a new predicted label's column.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.125.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
```

A frame containing the original frame's columns and a column with the predicted label

28.126 *Commands* model:svm/test

[ALPHA] Predict test frame labels and return metrics.

28.126.1 POST /v1/commands/

28.126.2 GET /v1/commands/:id

Request**Route**

```
POST /v1/commands/
```

Body

name model:svm/test

arguments **model** : <bound method AtkEntityType.__name__ of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
 <Missing Description>

frame : <bound method AtkEntityType.__name__ of
 <trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
 frame whose labels are to be predicted.

label_column : unicode

Column containing the actual label for each observation.

observation_columns : list (default=None)

Column(s) containing the observations whose labels are to be predicted and tested.
 Default is to test over the columns the SvmModel was trained on.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Predict the labels for a test frame and run classification metrics on predicted and target labels.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.126.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

dict

object

An object with binary classification metrics. The data returned is composed of multiple components:

```
<object>.accuracy : double <object>.confusion_matrix : table <object>.f_measure : double
<object>.precision : double <object>.recall : double
```

28.127 *Commands* model:svm/train

[ALPHA] Train SVM model based on another frame.

28.127.1 POST /v1/commands/

28.127.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model:svm/train

arguments **model** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

frame : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773090>>
A frame to train the model on.

label_column : unicode
Column name containing the label for each observation.

observation_columns : list
Column(s) containing the observations.

intercept : bool (default=None)
The algorithm adds an intercept. Default is true.

num_iterations : int32 (default=None)
Number of iterations. Default is 100.

step_size : int32 (default=None)
Step size for optimizer. Default is 1.0.

reg_type : unicode (default=None)
Regularization L1 or L2. Default is L2.

reg_param : float64 (default=None)
Regularization parameter. Default is 0.01.

mini_batch_fraction : float64 (default=None)
Mini batch fraction parameter. Default is 1.0.

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Description

Creating a SVM Model using the observation column and label column of the train frame.

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.127.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
_Unit
```

28.128 *Commands* model/rename

rename a model

28.128.1 POST /v1/commands/

28.128.2 GET /v1/commands/:id

Request

Route

```
POST /v1/commands/
```

Body

name model/rename

arguments **model** : <bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
<Missing Description>

new_name : unicode

The name to which the model will be renamed.

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Description

Response

Status

```
200 OK
```

Body

Returns information about the command. See the Response Body for Get Command here below. It is the same.

28.128.3 GET /v1/commands/:id

Request

Route

```
GET /v1/commands/18
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

```
<bound method AtkEntityType.__name__ of
<trustedanalytics.rest.jsonschema.AtkEntityType object at 0x7f6675773110>>
```

28.129 Command List

Command Name (explained here)	Description
frame/add_columns	Add columns to current frame.
frame/assign_sample	Randomly group rows into user-defined classes.
frame/bin_column	Classify data into user-defined groups.
frame/bin_column_equal_depth	Classify column into groups with the same frequency.
frame/bin_column_equal_width	Classify column into same-width groups.
frame/categorical_summary	Build summary of the data.
frame/classification_metrics	Model statistics of accuracy, precision, and others.
frame/column_median	Calculate the (weighted) median of a column.
frame/column_mode	Evaluate the weights assigned to rows.
frame/column_summary_statistics	Calculate multiple statistics for a column.
frame/compute_misplaced_score	
frame/copy	New frame with copied columns.
frame/correlation	Calculate correlation for two columns of current frame.
frame/correlation_matrix	Calculate correlation matrix for two or more columns.
frame/count_where	Counts qualified rows.
frame/covariance	Calculate covariance for exactly two columns.

Continued on next page

Table 28.1 – continued from previous page

Command Name (explained here)	Description
frame/covariance_matrix	Calculate covariance matrix for two or more columns.
frame/cumulative_percent	[BETA] Add column to frame with cumulative percent sum.
frame/cumulative_sum	[BETA] Add column to frame with cumulative percent sum.
frame/dot_product	[ALPHA] Calculate dot product for each row in current frame.
frame/drop_columns	Remove columns from the frame.
frame/drop_duplicates	Modify the current frame, removing duplicate rows.
frame/ecdf	Builds new frame with columns for data and distribution.
frame/entropy	Calculate the Shannon entropy of a column.
frame/export_to_csv	Write current frame to HDFS in csv format.
frame/export_to_hbase	Write current frame to HBase table.
frame/export_to_hive	Write current frame to Hive table.
frame/export_to_jdbc	Write current frame to Jdbc table.
frame/export_to_json	Write current frame to HDFS in JSON format.
frame/flatten_column	Spread data to multiple rows based on cell data.
frame/group_by	[BETA] Summarized Frame with Aggregations.
frame/histogram	[BETA] Compute the histogram for a column in a frame.
frame/loadhbase	Append data from an hBase table into an existing (possibly empty) FrameRDD
frame/loadhive	Append data from a hive table into an existing (possibly empty) frame
frame/loadjdbc	Append data from a Jdbc table into an existing (possibly empty) frame
frame/quantiles	New frame with Quantiles and their values.
frame/rename	Change the name of the current frame.
frame/sort	[BETA] Sort by one or more columns.
frame/sorted_k	[ALPHA] Get a sorted subset of the data.
frame/tally	[BETA] Count number of times a value is seen.
frame/tally_percent	[BETA] Compute a cumulative percent count.
frame/top_k	Most or least frequent column values.
frame/unflatten_column	Compacts data from multiple rows based on cell data.
frame:/filter	Select all rows which satisfy a predicate.
frame:/join	[BETA] Join two data frames (similar to SQL JOIN).
frame:/label_propagation	Label Propagation on Gaussian Random Fields.
frame:/load	Append data from a csv/xml into an existing (possibly empty) frame
frame:/loopy_belief_propagation	Message passing to infer state probabilities.
frame:/rename_columns	Rename columns
frame:edge/add_edges	Add edges to a graph.
frame:edge/rename_columns	Rename columns for edge frame.
frame:vertex/add_vertices	Add vertices to a graph.
frame:vertex/drop_duplicates	Remove duplicate vertex rows.
frame:vertex/filter	
frame:vertex/rename_columns	Rename columns for vertex frame.
graph/annotate_degrees	Make new graph with degrees.
graph/annotate_weighted_degrees	Calculates the weighted degree of each vertex with respect to an (optional) set of labels.
graph/clustering_coefficient	Coefficient of graph with respect to labels.
graph/copy	Make a copy of the current graph.
graph/graphx_connected_components	Implements the connected components computation on a graph by invoking graphx api.
graph/graphx_pagerank	Determine which vertices are the most important.
graph/graphx_triangle_count	Number of triangles among vertices of current graph.
graph/ml/belief_propagation	Classification on sparse data using Belief Propagation.
graph/rename	Rename a graph in the database.
graph:/define_edge_type	Define an edge type.

Continued on next page

Table 28.1 – continued from previous page

Command Name (explained here)	Description
graph:/define_vertex_type	Define a vertex type by label.
graph:/edge_count	Get the total number of edges in the graph.
graph:/export_to_titan	Convert current graph to TitanGraph.
graph:/ml/kclique_percolation	[ALPHA] Find groups of vertices with similar attributes.
graph:/vertex_count	Get the total number of vertices in the graph.
graph:titan/export_to_graph	Export from ta.TitanGraph to ta.Graph.
graph:titan/graph_clustering	Performs graph clustering over an initial titan graph.
graph:titan/query/gremlin	Executes a Gremlin query.
graph:titan/vertex_sample	Make subgraph from vertex sampling.
model/rename	rename a model
model:collaborative_filtering/new	Collaborative filtering recommend model.
model:collaborative_filtering/recommend	[BETA] Collaborative filtering (als/cgd) model
model:collaborative_filtering/train	Collaborative filtering (als/cgd) model
model:k_means/new	create a new model
model:k_means/predict	[BETA] Predict the cluster assignments for the data points.
model:k_means/publish	[BETA] Creates a tar file that will used as input to the scoring engine
model:k_means/train	[BETA] Creates KMeans Model from train frame.
model:lda/new	Creates Latent Dirichlet Allocation model
model:lda/predict	[BETA] Predict conditional probabilities of topics given document.
model:lda/publish	[BETA] Creates a tar file that will used as input to the scoring engine
model:lda/train	[BETA] Creates Latent Dirichlet Allocation model
model:libsvm/new	[ALPHA] model:libsvm/new
model:libsvm/predict	[ALPHA] New frame with new predicted label column.
model:libsvm/publish	[BETA] Creates a tar file that will used as input to the scoring engine
model:libsvm/score	[ALPHA] Calculate the prediction label for a single observation.
model:libsvm/test	[ALPHA] Predict test frame labels and return metrics.
model:libsvm/train	[ALPHA] Train Lib Svm model based on another frame.
model:linear_regression/new	
model:linear_regression/predict	[ALPHA] Make new frame with column for label prediction.
model:linear_regression/train	[ALPHA] Build linear regression model.
model:logistic_regression/new	Create a ‘new’ instance of logistic regression model.
model:logistic_regression/predict	[ALPHA] Make a new frame with a column for label prediction.
model:logistic_regression/test	[ALPHA] Predict test frame labels and show metrics.
model:logistic_regression/train	[ALPHA] Build logistic regression model.
model:naive_bayes/new	create a new model
model:naive_bayes/predict	[ALPHA] Predict
model:naive_bayes/train	[ALPHA] Build a naive bayes model.
model:principal_components/new	Create a ‘new’ instance of principal component model.
model:principal_components/predict	[ALPHA] Predict using principal components model.
model:principal_components/publish	[BETA] Creates a tar file that will be used as input to the scoring engine
model:principal_components/train	Build principal components model.
model:random_forest_classifier/new	Create a ‘new’ instance of random forest classifier model.
model:random_forest_classifier/predict	[ALPHA] Predict the labels for the data points.
model:random_forest_classifier/publish	[BETA] Creates a tar file that will be used as input to the scoring engine
model:random_forest_classifier/test	[ALPHA] Predict test frame labels and return metrics.
model:random_forest_classifier/train	[ALPHA] Build Random Forests Classifier model.
model:random_forest_regressor/new	<Missing Doc>
model:random_forest_regressor/predict	[ALPHA] Predict the values for the data points.
model:random_forest_regressor/publish	[BETA] Creates a tar file that will be used as input to the scoring engine

Continued on next page

Table 28.1 – continued from previous page

Command Name (explained here)	Description
<code>model:random_forest_regressor/train</code>	[ALPHA] Build Random Forests Regressor model.
<code>model:svm/new</code>	[ALPHA] create a new model
<code>model:svm/predict</code>	[ALPHA] Make new frame with additional column for predicted label.
<code>model:svm/test</code>	[ALPHA] Predict test frame labels and return metrics.
<code>model:svm/train</code>	[ALPHA] Train SVM model based on another frame.

REST API ENTITIES

29.1 *Entities* Create Entity

Creates a new entity, like a frame, graph, or model.

29.1.1 POST /v1/:entities/

Request

Route

```
POST /v1/frames/  
POST /v1/graphs/  
POST /v1/models/
```

Body

Name	Description	Default	Valid Values	Example Values
name	name for the entity	null	alphanumeric UTF-8 strings	'weather_frame1'

```
{  
  "name": "weather_frame1"  
}
```

Headers

```
Authorization: test_api_key_1  
Content-type: application/json
```

Response

Status

```
200 OK
```

Body

Returns a summary of the entity.

Name	Description
uri	entity id (engine-assigned)
name	entity name (user-assigned)
links	links to the entity
entity_type	e.g. "frame:", "frame:vertex", "graph:"
status	status: Active, Deleted, Deleted_Final

Extra fields specific to frames:

Name	Description
schema	frame schema info columns: [(name, type)]
row_count	number of rows in the frame

```
{
  "name": "weather_frame1",
  "uri": "frames/8",
  "schema": {
    "columns": []
  },
  "row_count": 0,
  "links": [
    {
      "rel": "self",
      "uri": "http://localhost:9099/v1/frames/8",
      "method": "GET"
    }
  ],
  "entity_type": "frame:",
  "status": "Active"
}
```

Headers

```
Content-Length: 279
Content-Type: application/json; charset=UTF-8
Date: Thu, 14 May 2015 23:42:27 GMT
Server: spray-can/1.3.1
build_id: TheReneNumber
```

29.2 Entities Drop Entity

Deletes an entity

29.2.1 DELETE /v1/:entities/:id

Request

Route

```
DELETE /v1/frames/25
DELETE /v1/graphs/6
DELETE /v1/models/4
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

200 OK

Body

(None)

29.3 *Entities* Get Entity

Gets information about specific entity, like a frame, graph, or model. There are two options: get by id or get by name.

29.3.1 GET /v1/:entities/:id

29.3.2 GET /v1/:entities?name=

Request

Route

```
GET /v1/frames/3
GET /v1/graphs/1
GET /v1/models/4
GET /v1/frames?name=weather_frame1
GET /v1/graphs?name=networkB
```

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
```

Response

Status

200 OK

Body

Returns information about the entity

Name	Description
uri	entity id (engine-assigned)
name	entity name (user-assigned)
links	links to the entity
entity_type	e.g. "frame:", "frame:vertex", "graph:"
status	status: Active, Deleted, Deleted_Final

Extra fields specific to frames:

Name	Description
schema	frame schema info columns: [(name, type)]
row_count	number of rows in the frame

```
{
  "uri": "frames/7",
  "name": "super_frame",
  "entity_type": "frame:",
  "status": "Active"
  "links": [{
    "rel": "self",
    "uri": "http://localhost:9099/v1/frames/7",
    "method": "GET"
  }],
  "schema": {
    "columns": [{
      "name": "c",
      "data_type": "int32",
      "index": 0
    }, {
      "name": "twice",
      "data_type": "int32",
      "index": 1
    }, {
      "name": "sample_bin",
      "data_type": "string",
      "index": 2
    }, {
      "name": "c_binned",
      "data_type": "int32",
```



```

    "index": 3
  }
},
"row_count": 8675309,
}

```

Headers

```

Content-Length: 279
Content-Type: application/json; charset=UTF-8
Date: Thu, 14 May 2015 23:42:27 GMT
Server: spray-can/1.3.1
build_id: TheReneNumber

```

29.4 *Entities* Get Named Entities

Gets list of short entries for all named entities in the entity collection.

29.4.1 GET /v1/entities

Request**Route**

```

GET /v1/frames
GET /v1/graphs
GET /v1/models

```

Body

(None)

Headers

```

Authorization: test_api_key_1
Content-type: application/json

```

Response**Status**

200 OK

Body

Returns a list of entity entries for the given collection, where an entry is defined as...

Name	Description
id	entity id (engine-assigned)
name	entity name (user-assigned)
url	url to the entity
entity_type	e.g. “frame:”, “frame:vertex”, “graph:”, “model:kmeans”

Example for GET /v1/frames:

```
[
  {
    "id": 7,
    "name": "super_frame",
    "url": "http://localhost:9099/v1/frames/7",
    "entity_type": "frame:"
  },
  {
    "id": 8,
    "name": "weather_frame1",
    "url": "http://localhost:9099/v1/frames/8",
    "entity_type": "frame:"
  }
]
```

Headers:

```
Content-Length: 279
Content-Type: application/json; charset=UTF-8
Date: Thu, 14 May 2015 23:42:27 GMT
```

29.5 Entities Get Frame Data

Gets data from a frame by rows.

29.5.1 GET /v1/frames/:id/data?offset=:offset&count=:count

Request**Route**

```
GET /v1/frames/7/data?offset=0&count=10
```

Parameters

offset: index of the starting row

count: number of rows to retrieve

Body

(None)

Headers

```
Authorization: test_api_key_1
Content-type: application/json
Accept: application/json,text/plain
```

200 OK

Returns rows of data

Name	Description
name	name of the operation “getRows”
complete	boolean indicating completion of data fetch
result	data: list of rows of data
	schema: row structure (column names and data types)

```
{
  "name": "getRows",
  "complete": true,
  "result": {
    "data": [[1, 2, "validate", 0], [2, 4, "validate", 0], [3, 6, "validate", 1], [2, 4, "validate", 1]],
    "schema": {
      "columns": [{
        "name": "c",
        "data_type": "int32",
        "index": 0
      }, {
        "name": "twice",
        "data_type": "int32",
        "index": 1
      }, {
        "name": "sample_bin",
        "data_type": "string",
        "index": 2
      }, {
        "name": "c_binned",
        "data_type": "int32",
        "index": 3
      }
    ]
  }
},
}
```

```
Content-Length: 753
Content-Type: application/json; charset=UTF-8
Date: Thu, 14 May 2015 23:42:27 GMT
```

The data is considered ‘unstructured’, therefore taking a certain number of rows, the rows obtained may be different every time the command is executed, even if the parameters do not change.

[Need a note in here about performance and recommended usage]

REST API INFO

Basic server information supplying API versions.

30.1 GET /info

30.1.1 Request

Route

GET /info

Body

(None)

Headers

(None)

30.1.2 Response

Status

200 OK

Body

Returns server information and API versions.

<pre>{ "name": "Trusted Analytics", "identifier": "ia", "versions": ["v1"] }</pre>
--

Headers

```
Content-Length: 75
Content-Type: application/json; charset=UTF-8
Date: Thu, 14 May 2015 23:42:27 GMT
```

Part VII

References

- [Glossary](#)
- [Legal Statement](#)
- [Index](#)
- [Appendices](#)
- [Errata](#)
- [PDF](#)

GLOSSARY

Adjacency List A representation of a graph as a list. Each line of the list consists of a unique vertex identification, and a list of all of that vertex's neighboring vertices.

Example:

Node	Connection List
/-----/	
A	B, D
B	A, C, D
C	B
D	A, B

Aggregation Function A mathematical function which is usually computed over a single column. Supported functions:

- avg : The average (mean) value in the column
- count : The count of the rows
- count_distinct : The count of unique rows
- max : The largest (most positive) value in the column
- min : The least (most negative) value in the column
- stdev : The standard deviation of the values in the column, see [Wikipedia: Standard Deviation](http://en.wikipedia.org/wiki/Standard_deviation)¹
- sum : The result of adding all the values in the column together
- var : The variance of the values in the column, see [Wikipedia: Variance](https://en.wikipedia.org/wiki/Variance)² and *Bias vs Variance*

Alpha See *API Maturity Tags*.

Alternating Least Squares A method used in some approaches to multidimensional scaling, where a goodness-of-fit measure for some data is minimized in a series of steps, each involving the application of the *least squares* method of parameter estimation.

See the API section on the *Collaborative Filter Model* for an in-depth discussion of this method.

API Maturity Tags Functions in the API may be at different levels of software maturity. Where a function is not mature, the documentation will note it with one of the following tags. The absence of a tag means the function is standardized and fully tested.

[ALPHA] Indicates a function or feature which has been developed, but has not been completely tested. Use this function with caution. This function may be changed or eliminated in future releases.

¹http://en.wikipedia.org/wiki/Standard_deviation

²<https://en.wikipedia.org/wiki/Variance>

[BETA] Indicates a function or feature which has been developed and preliminarily tested, but has not been completely tested. Use this function with caution. This function may be changed in future releases.

[DEPRECATED] Indicates a function or feature which is no longer supported. It is recommended that an alternate solution be found. This function may be removed in future releases.

ASCII Abbreviated from American Standard Code for Information Interchange, ASCII is a character-encoding scheme. Originally based on the English alphabet, it encodes 128 specified characters into 7-bit binary integers.

Average Path Length In network topology, the average number of steps along the shortest paths for all possible pairs of vertices.

Bayesian Inference A probabilistic graphical model representing the conditional dependencies amongst a set of random variables with a directed acyclic graph.

Contrast with *Markov Random Fields*

For more information, see [Wikipedia: Bayesian Network](#)³.

Belief Propagation See *Loopy Belief Propagation*.

Beta See *API Maturity Tags*.

Bias vs Variance In this context, “bias” means accuracy, while “variance” means accounting for outlier data points.

Bias-variance tradeoff In supervised classifier training, the problem of minimizing two sources of prediction error: erroneous assumptions in the learning algorithm, and sensitivity to small details in the training data (in other words, over-fitting) when generalizing to a testing data set.

Central Tendency A typical value for a probability distribution. It may also be called a center or location of the distribution. Colloquially, measures of central tendency are often called averages.

Centrality From [Wikipedia: Centrality](#)⁴:

In graph theory and network analysis, centrality of a vertex measures its relative importance within a graph. Applications include how influential a person is within a social network, how important a room is within a building (space syntax), and how well-used a road is within an urban network. There are four main measures of centrality: degree, betweenness, closeness, and eigenvector. Centrality concepts were first developed in social network analysis, and many of the terms used to measure centrality reflect their sociological origin.⁵

Centrality (Katz) See *Katz Centrality*.

Centrality (PageRank) See *Centrality*.

Character-Separated Values A file containing tabular data (numbers and text) in plain-text form. The file can consist of any number of records, separated by a unique character. New line characters are usually used for this purpose. Each record consists of one or more fields, separated by some unique character. Commas are usually used for this purpose. Tab characters are also quite common.

Classification The process of predicting category membership for a set of observations based on a model learned from the known categorical groupings of another set of observations.

Clustering See *Collaborative Clustering*.

Collaborative Clustering The unsupervised grouping of observations based on one or more character traits.

Collaborative Filtering The process of filtering for information or patterns using techniques involving collaboration among multiple agents, viewpoints, data sources, etc.⁶

³http://en.wikipedia.org/wiki/Bayesian_network

⁴<http://en.wikipedia.org/wiki/Centrality>

⁵ Newman, M.E.J. 2010. Networks: An Introduction. Oxford, UK: Oxford University Press.

⁶ Terveen, Loren; Hill, Will (2001). Beyond Recommender Systems: Helping People Help Each Other pp. 6. Addison-Wesley.

Comma-Separated Variables See *Character-Separated Values*.

Community Structure Detection For complex networks, the process of identifying vertices that can be easily grouped into densely-connected sub-groupings.

Confusion Matrices Plural form of *Confusion Matrix*

Confusion Matrix In machine learning, a table describing the performance of a supervised classification algorithm, in which each column corresponds to instances of a predicted class, while each row represents the instances of the true class. Also known as contingency table, error matrix, or misclassification matrix.

Conjugate Gradient Descent Trusted Analytics implements this algorithm. Specifically, it uses CGD with bias for collaborative filtering.

For more information: [Factorization Meets the Neighborhood \(pdf\)](#)⁷ (see equation 5).

Connected Component In graph theory, a sub-graph in which any two vertices are interconnected but share no connections with other vertices in the sub-graph.

Convergence Where a calculation (often an iterative calculation) reaches a certain value.

For more information see: [Wikipedia: Convergence \(mathematics\)](#)⁸.

CSV See *Character-Separated Values*

Degree The degree of a vertex is the number of edges incident to the vertex. Loops are counted twice. The maximum and minimum degree of a graph are the maximum and minimum degree of its vertices.

For more information see: [Wikipedia: Degree \(graph theory\)](#)⁹.

Deprecated See *API Maturity Tags*.

Directed Acyclic Graph (DAG) In mathematics and computer science, a graph formed by a collection of vertices and directed edges, each edge connecting one vertex to another, such that there is no way to start at some vertex v and follow a sequence of edges that eventually loops back to v again.

Contrast with *Undirected Graph*.

See [Wikipedia: Directed Acyclic Graph](#)¹⁰.

ECDF See *Empirical Cumulative Distribution*

Edge A connection — either directed or not — between two vertices in a graph.

Empirical Cumulative Distribution $\hat{F}_n(t)$ is a step function with jumps i/n at observation values, where i is the number of tied observations at that value. Missing values are ignored.

For observations $x = (x_1, x_2, \dots, x_n)$, $\hat{F}_n(t)$ is the fraction of observations less than or equal to t .

$$\hat{F}_n(t) = \frac{x_i \leq t}{n} = \frac{1}{n} \sum_{i=1}^n \text{Indicator}\{x_i \leq t\}.$$

where $\text{Indicator}\{A\}$ is the indicator of event A . For a fixed t , the indicator $\text{Indicator}\{x_i \leq t\}$ is a Bernoulli random variable with parameter $p = F(t)$, hence $n\hat{F}_n(t)$ is a binomial random variable with mean $nF(t)$ and variance $nF(t)(1 - F(t))$. This implies that $\hat{F}_n(t)$ is an unbiased estimator for $F(t)$.

Enumerate Verb — To specify each member of a sequence individually in incrementing order.

Equal Depth Binning Equal depth binning places column values into groups such that each group contains the same number of elements.

⁷<http://public.research.att.com/~volinsky/netflix/kdd08koren.pdf>

⁸[http://en.wikipedia.org/wiki/Convergence_\(mathematics\)](http://en.wikipedia.org/wiki/Convergence_(mathematics))

⁹[https://en.wikipedia.org/wiki/Degree_\(graph_theory\)](https://en.wikipedia.org/wiki/Degree_(graph_theory))

¹⁰https://en.wikipedia.org/wiki/Directed_acyclic_graph

Equal Width Binning Equal width binning places column values into groups such that the values in each group fall within the same interval and the interval width for each group is equal.

Extract, Transform, and Load From [Wikipedia: Extract, Transform, and Load](#)¹¹:

In computing, ETL (extract, transform, and load) refers to a process in database usage and especially in data warehousing that:

- Extracts data from outside sources
- Transforms it to fit operational needs, which can include quality levels
- Loads it into the end target (database, more specifically, operational data store, data mart, or data warehouse)

ETL systems are commonly used to integrate data from multiple applications, typically developed and supported by different vendors or hosted on separate computer hardware. The disparate systems containing the original data are frequently managed and operated by different employees. For example a cost accounting system may combine data from payroll, sales and purchasing.

F-Measure In machine learning, a metric that quantifies a classifier's accuracy. Traditionally defined as the harmonic mean of precision and recall. Also known as F1 score.

F-Score See *F-Measure*.

F1 Score See *F-Measure*.

float32 A real number with 32 bits of precision.

float64 A real number with 64 bits of precision.

Frame (capital F) A class object with the functionality to manipulate the data in a *frame* (lower case f).

frame (lower case f) A table database with rows and columns containing data.

GaBP See *Gaussian Belief Propagation*.

Gaussian Belief Propagation A special case of belief propagation when the underlying distributions are *Gaussian* (Weiss & Freeman¹²).

Gaussian Distribution, Normal Distribution A group of values, where the probability of any specific value:

- will fall between two real limits,
- is evenly centered around the mean,
- approaches zero on either side of the mean.

A Gaussian distribution is defined as:

$$f(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-i\frac{(x-i\mu)^2}{2i\sigma^2}}$$

- μ is the mean of the distribution.
- σ is the standard deviation.

Gaussian Random Fields A random group of vertices displaying a *Gaussian distribution* of one or more sets of properties.

Global Clustering Coefficient The global clustering coefficient is based on triplets of vertices. A triplet consists of three vertices that are connected by either two (open triplet) or three (closed triplet) undirected edges. A

¹¹http://en.wikipedia.org/wiki/Extract_transform_load

¹² Weiss, Yair; Freeman, William T. (October 2001). "Correctness of Belief Propagation in Gaussian Graphical Models of Arbitrary Topology". *Neural Computation* 13 (10): 2173|EM|2200. doi:10.1162/089976601750541769. PMID 11570995.

triangle consists of three closed triplets, one centered on each of the vertices. The global clustering coefficient is the number of closed triplets (or 3 x triangles) over the total number of triplets (both open and closed).

For more information see: [Wikipedia: Global Clustering Coefficient](#)¹³.

See also *Local Clustering Coefficient*.

Graph A representation of a set of vertices, where some pairs of objects are connected by edges. The links that connect some pairs of vertices are called edges. Typically, a graph is depicted in diagrammatic form as a set of dots for the vertices, joined by lines or curves for the edges. Graphs are one of the objects of study in discrete mathematics.

For more information see: [Wikipedia: Graph \(mathematics\)](#)¹⁴.

Graph Analytics The broad category of methods used to examine the statistical and structural properties of a graph, including:

1. Traversals – Algorithmic walk throughs of the graph to determine optimal paths and relationship between vertices.
2. Statistics – Important attributes of the graph such as degrees of separation, number of triangular counts, centralities (highly influential nodes), and so on.

Some are user-guided interactions, where the user navigates through the data connections, others are algorithmic, where a result is calculated by the software.

Graph learning is a class of graph analytics applying machine learning and data mining algorithms to graph data. This means that calculations are iterated across the nodes of the graph to uncover patterns and relationships. Thus, finding similarities based on relationships, or recursively optimizing some parameter across nodes.

For more information, see the article [Graph Analytics](#)¹⁵ by Pak Chung Wong.

Graph Database Directions As a shorthand, graph database terminology uses relative directions, assumed to be from whatever vertex you are currently using. These directions are:

- **left:** The calling frame's index
- **right:** The input frame's index
- **inner:** An intersection of indexes

So a direction like this: “The suffix to use from the left frame's overlapping columns” means to use the suffix from the calling frame's index.

Graph Element A graph element is an object that can have any number of key-value pairs, that is, properties, associated with it. Each element can have zero properties as well.

Gremlin A graph query language. Gremlin works with the Titan Graph Database, though it is made by a different company. For more information see: [Gremlin Wiki](#)¹⁶.

HBase Apache HBase is the Hadoop database, a distributed, scalable, big data store.

int32 An integer is a member of the set of positive whole numbers {1, 2, 3, . . . }, negative whole numbers {-1, -2, -3, . . . }, and zero {0}. Since a computer is limited, the computer representation of it can have 32 bits of precision.

int64 An integer is a member of the set of positive whole numbers {1, 2, 3, . . . }, negative whole numbers {-1, -2, -3, . . . }, and zero {0}. Since a computer is limited, the computer representation of it can have 64 bits of precision.

¹³https://en.wikipedia.org/wiki/Clustering_coefficient#Global_clustering_coefficient

¹⁴[http://en.wikipedia.org/wiki/Graph_\(mathematics\)](http://en.wikipedia.org/wiki/Graph_(mathematics))

¹⁵<http://vacommunity.org/article26>

¹⁶<https://github.com/tinkerpop/gremlin/wiki>

Ising Smoothing Parameter The smoothing parameter in the Ising model. For more information see: [Wikipedia: Ising Model](#)¹⁷.

You can use any positive float number, so 3, 2.5, 1, or 0.7 are all valid values. A larger smoothing value implies stronger relationships between adjacent random variables in the graph.

JSON Data in the JavaScript Object Notation format. An open standard format that uses human-readable text to transmit data objects consisting of attributevalue pairs. For more information see '<http://json.org>'.

K-S (Kolmogorov-Smirnov) Test From [Wikipedia: Kolmogorov-Smirnov Test](#)¹⁸:

In statistics, the K-S test is a nonparametric test of the equality of continuous, one-dimensional probability distributions that can be used to compare a sample with a reference probability distribution (one-sample K-S test), or to compare two samples (two-sample K-S test). The K-S statistic quantifies a distance between the empirical distribution function of the sample and the cumulative distribution function of the reference distribution, or between the empirical distribution functions of two samples.

Katz Centrality From [Wikipedia: Katz Centrality](#)¹⁹:

In Social Network Analysis (SNA) there are various measures of *centrality* which determine the relative importance of an actor (or node) within the network. Katz centrality was introduced by Leo Katz in 1953 and is used to measure the degree of influence of an actor in a social network.²⁰

Unlike typical centrality measures which consider only the shortest path (the geodesic) between a pair of actors, Katz centrality measures influence by taking into account the total number of walks between a pair of actors.²¹

Label Propagation A multi-pass process for grouping vertices.

See [Label Propagation \(LP\)](#).

For additional reference: [Learning from Labeled and Unlabeled Data with Label Propagation](#)²².

Labeled Data vs Unlabeled Data From [Wikipedia: Machine Learning / Algorithm Types](#)²³:

Supervised learning algorithms are trained on labeled examples, in other words, input where the desired output is known. While Unsupervised learning algorithms operate on unlabeled examples, in other words, input where the desired output is unknown.

Many machine-learning researchers have found that unlabeled data, when used in conjunction with a small amount of labeled data, can produce considerable improvement in learning accuracy.

For more information see: [Wikipedia: Semi-Supervised Learning](#)²⁴.

Lambda Adapted from: [Stanford: Machine Learning](#)²⁵:

This is the tradeoff parameter, used in [Label Propagation](#) on [Gaussian Random Fields](#). The regularization parameter is a control on fitting parameters. It is used in machine learning algorithms to prevent overfitting. As the magnitude of the fitting parameter increases, there will be an increasing penalty on the cost function. This penalty is dependent on the squares of the parameters as well as the magnitude of lambda.

¹⁷http://en.wikipedia.org/wiki/Ising_model

¹⁸http://en.wikipedia.org/wiki/K-S_Test

¹⁹http://en.wikipedia.org/wiki/Katz_centrality

²⁰ Katz, L. (1953). A New Status Index Derived from Sociometric Index. *Psychometrika*, 39-43.

²¹ Hanneman, R. A., & Riddle, M. (2005). *Introduction to Social Network Methods* (<http://faculty.ucr.edu/hanneman/nettext/>).

²²<http://lvk.cs.msu.su/bruzz/articles/classification/zhu02learning.pdf>

²³http://en.wikipedia.org/wiki/Machine_learning#Algorithm_types

²⁴http://en.wikipedia.org/wiki/Semi-supervised_learning

²⁵<http://openclassroom.stanford.edu/MainFolder/DocumentPage.php?course=MachineLearning&doc=exercises/ex5/ex5.html>

Lambda Function An anonymous function or function literal in code. Lambda functions are used when a method requires a function as an input parameter and the function is coded directly in the method call.

Further examples and explanations can be found at this page: [Python User Functions](#).

Related term: *Python User-defined Function*.

Warning: This term is often used where a *Python user-defined function* is more accurate. A key distinction is that the lambda function is not referable by a name.

Latent Dirichlet Allocation From [Wikipedia: Latent Dirichlet Allocation](#)²⁶:

[A] generative model that allows sets of observations to be explained by unobserved groups that explain why some parts of the data are similar. For example, if observations are words collected into documents, it posits that each document is a mixture of a small number of topics and that each word’s creation is attributable to one of the document’s topics. LDA is an example of a topic model and was first presented as a graphical model for topic discovery by David Blei, Andrew Ng, and Michael Jordan in 2003.

Least Squares A mathematical procedure for finding the best-fitting curve to a given set of points by minimizing the sum of the squares of the offsets (“the residuals”) of the points from the curve. The sum of the squares of the offsets is used instead of the offset absolute values because this allows the residuals to be treated as a continuous differentiable quantity. However, because squares of the offsets are used, outlying points can have a disproportionate effect on the fit, a property which may or may not be desirable depending on the problem at hand.

LineFile A data format where the records are line-delimited.

Local Clustering Coefficient The local clustering coefficient of a vertex in a graph quantifies how close its neighbors are to being a clique (complete graph).

For more information see: [Wikipedia: Local Clustering Coefficient](#)²⁷.

See also *Global Clustering Coefficient*.

Loopy Belief Propagation Belief Propagation is an algorithm that makes inferences on graph models, like a *Bayesian inference* or *Markov Random Fields*. It is called Loopy when the algorithm runs iteratively until convergence.

For more information see: [Wikipedia: Belief Propagation](#)²⁸.

Machine Learning Machine learning is a branch of artificial intelligence. It is about constructing and studying software that can “learn” from data. The more iterations the software computes, the better it gets at making that calculation. For more information, see [Wikipedia](#)²⁹.

MapReduce MapReduce is a programming model for processing large data sets with a parallel, distributed algorithm on a cluster. It is composed of a map() procedure that performs filtering and sorting (such as sorting students by first name into queues, one queue for each name) and a reduce() procedure that performs a summary operation (such as counting the number of students in each queue, yielding name frequencies). The “MapReduce System” (also called “infrastructure” or “framework”) orchestrates by marshaling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing for redundancy and fault tolerance.

For more information see: [Wikipedia: MapReduce](#)³⁰.

²⁶http://en.wikipedia.org/wiki/Latent_Dirichlet_allocation

²⁷https://en.wikipedia.org/wiki/Clustering_coefficient#Local_clustering_coefficient

²⁸http://en.wikipedia.org/wiki/Loopy_belief_propagation

²⁹https://en.wikipedia.org/wiki/Machine_learning

³⁰http://en.wikipedia.org/wiki/Map_reduce

Markov Random Fields Markov Random fields, or Markov Network, are an undirected graph model that may be cyclic. This contrasts with *Bayesian inference*, which is directed and acyclic.

For more information see: [Wikipedia: Markov Random Field](#)³¹.

OLAP Online analytical processing. An approach to answering MDA (Multi-Dimensional Analytical) queries swiftly. The term OLAP (OnLine Analytical Processing) was created as a slight modification of the traditional database term OLAP.

For more information see: [Wikipedia: Online analytical processing](#)³².

OLTP Online transaction processing. A class of information systems that facilitate and manage transaction-oriented applications. OLAP involves gathering input information, processing the information and updating existing information to reflect the gathered and processed information.

For more information see: [Wikipedia: Online transaction processing](#)³³.

PageRank An algorithm to measure the importance of vertices.

PageRank works by counting the number and quality of edges to a vertex to determine a rough estimate of how important the vertex is. The underlying assumption is that more important vertices are likely to have more edges from other vertices.

For more information see: [Wikipedia: PageRank](#)³⁴.

PageRank Centrality See *Centrality*.

Precision/Recall From [Wikipedia: Precision and Recall](#)³⁵:

In pattern recognition and information retrieval with binary classification, precision (also called positive predictive value) is the fraction of retrieved instances that are relevant, while recall (also known as sensitivity) is the fraction of relevant instances that are retrieved. Both precision and recall are therefore based on an understanding and measure of relevance.

Property Map A property map is a key-value map. Both edges and vertices have property maps.

For more information see: [Tinkerpop: Property Graph Model](#)³⁶.

Python User-defined Function A Python User-defined Function (UDF) is a Python function written by the user on the client-side which can execute in a distributed fashion on the cluster. For further explanation, see [Python User Functions](#)

Further examples and explanations can be found at [Python User Functions](#).

Related: *Lambda Function*.

Quantile One value of a set that partitions a collection of data. Each partition (also known as a quantile) contains all the collection elements from the given value, up to (but not including) the lowest value of the next quantile.

RDF See *Resource Description Framework*

Receiver Operating Characteristic From [Wikipedia: Receiver Operating Characteristic](#)³⁷:

In signal detection theory, a receiver operating characteristic (ROC), or simply ROC curve, is a graphical plot which illustrates the performance of a binary classifier system as its discrimination threshold is varied. It is created by plotting the fraction of true positives out of the total actual positives (TPR = true positive rate) vs. the fraction of false positives out of the total actual negatives

³¹http://en.wikipedia.org/wiki/Markov_random_field

³²https://en.wikipedia.org/wiki/Online_analytical_processing

³³https://en.wikipedia.org/wiki/Online_transaction_processing

³⁴<http://en.wikipedia.org/wiki/PageRank>

³⁵http://en.wikipedia.org/wiki/Precision_and_recall

³⁶<https://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model>

³⁷https://en.wikipedia.org/wiki/Receiver_operating_characteristic

(FPR = false positive rate), at various threshold settings. TPR is also known as sensitivity or recall in machine learning. The FPR is also known as the fall-out and can be calculated as one minus the more well known specificity. The ROC curve is then the sensitivity as a function of fall-out. In general, if both of the probability distributions for detection and false alarm are known, the ROC curve can be generated by plotting the Cumulative Distribution Function (area under the probability distribution from $-\infty$ to $+\infty$) of the detection probability in the y-axis versus the Cumulative Distribution Function of the false alarm probability in x-axis.

Recommendation Systems From [Wikipedia: Recommender System](#)³⁸:

Recommender systems or recommendation systems (sometimes replacing “system” with a synonym such as platform or engine) are a subclass of information filtering system that seek to predict the ‘rating’ or ‘preference’ that user would give to an item^{39 40}.

Resource Description Framework A specific format for storing graphs. Vertices also referred to as resources, have property/value pairs describing the resource. A vertex is any object which can be pointed to by a URI. Properties are attributes of the vertex, and values are either specific values for the attribute, or the URI for another vertex. For example, information in a particular vertex, might include the property “Author”. The value for the Author property could be either a string giving the name of the author, or a link to another resource describing the author. Sets of properties are defined within RDF Vocabularies (or schemas). A vertex may include properties defined in different schemas. The properties within a resource description are associated with a certain schema definition using the XML namespace mechanism.

ROC See [Receiver Operating Characteristic](#)

Row Functions Refer to [Lambda Function](#) and [Python User-defined Function](#)

Schema A computer structure that defines the structure of something else.

Semi-Supervised Learning In Semi-Supervised learning algorithms, most the input data are not labeled and a small amount are labeled. The expectation is that the software “learns” to calculate faster than in either supervised or unsupervised algorithms.

For more information see: [Supervised Learning](#), and [Unsupervised Learning](#).

Simple Random Sampling In statistics, a simple random sample (SRS) is a subset of individuals (a sample) chosen from a larger set (a population). Each individual is chosen randomly and entirely by chance, such that each individual has the same probability of being chosen at any stage during the sampling process, and each subset of k individuals has the same probability of being chosen for the sample as any other subset of k individuals⁴¹. This process and technique is known as simple random sampling. A simple random sample is an unbiased surveying technique.

For more information see: [Wikipedia: Simple Random Sample](#)⁴².

Smoothing Smoothing means to reduce the “noise” in a data set. “In smoothing, the data points of a signal are modified so individual points (presumably because of noise) are reduced, and points that are lower than the adjacent points are increased leading to a smoother signal.”

For more information see:

[Wikipedia: Smoothing](#)⁴³

[Wikipedia: Relaxation \(iterative method\)](#)⁴⁴

³⁸http://en.wikipedia.org/wiki/Recommendation_system

³⁹ Francesco Ricci and Lior Rokach and Bracha Shapira (2011). Recommender Systems Handbook, pp. 1-35. Springer.

⁴⁰ Lev Grossman (2010). *How Computers Know What We Want IEMI Before We Do* (<http://content.time.com/time/magazine/article/0,9171,1992403,00.html>). Time.

⁴¹ Yates, Daniel S.; David S. Moore, Daren S. Starnes (2008). The Practice of Statistics, 3rd Ed. Freeman. ISBN 978-0-7167-7309-2.

⁴²https://en.wikipedia.org/wiki/Simple_random_sampling

⁴³<http://en.wikipedia.org/wiki/Smoothing>

⁴⁴[http://en.wikipedia.org/wiki/Relaxation_\(iterative_method\)](http://en.wikipedia.org/wiki/Relaxation_(iterative_method))

Stratified Sampling In statistics, stratified sampling is a method of sampling from a population. In statistical surveys, when subpopulations within an overall population vary, it is advantageous to sample each subpopulation (stratum) independently. Stratification is the process of dividing members of the population into homogeneous subgroups before sampling. The strata should be mutually exclusive: every element in the population must be assigned to only one stratum. The strata should also be collectively exhaustive: no population element can be excluded. Then simple random sampling or systematic sampling is applied within each stratum. This often improves the representativeness of the sample by reducing sampling error. It can produce a weighted mean that has less variability than the arithmetic mean of a simple random sample of the population.

For more information see: [Wikipedia: Stratified Sampling](#)⁴⁵.

Superstep, Supersteps A single iteration of an algorithm.

Supervised Learning Supervised learning refers to algorithms where the input data are all labeled, and the outcome of the calculation is known. These algorithms train the software to make a certain calculation.

For more information see: [Unsupervised Learning](#), and [Semi-Supervised Learning](#).

Tab-Separated Variables See [Character-Separated Values](#).

TitanGraph A class object with the functionality to manipulate the data in a *graph*.

Topic Modeling Topic models provide a simple way to analyze large volumes of unlabeled text. A “topic” consists of a cluster of words that frequently occur together. Using contextual clues, topic models can connect words with similar meanings and distinguish between uses of words with multiple meanings.

Transaction Processing From [Wikipedia: Transaction Processing](#)⁴⁶:

In computer science, transaction processing is information processing that is divided into individual, indivisible operations, called transactions. Each transaction must succeed or fail as a complete unit; it cannot be only partially complete.

Transactional Functionality See [Transaction Processing](#).

UDF See [Python User-defined Function](#).

Undirected Graph An undirected graph is one in which the edges have no orientation (direction). The edge (a, b) is identical to the edge (b, a), in other words, they are not ordered pairs, but sets {u, v} (or 2-multisets) of vertices. The maximum number of edges in an undirected graph without a self-loop is $\frac{n(n-1)}{2}$

Contrast with [Directed Acyclic Graph \(DAG\)](#).

For more information see: [Wikipedia: Undirected Graph](#)⁴⁷.

Unicode A data type consisting of a string of characters designed to represent all characters in the world, a universal character set.

Unsupervised Learning Unsupervised learning refers to algorithms where the input data are not labeled, and the outcome of the calculation is unknown. In this case, the software needs to “learn” how to make the calculation.

For more information see: [Supervised Learning](#), and [Semi-Supervised Learning](#).

Vertex A vertex is an object in a graph. Each vertex has an ID and a property map. In Giraph, a long integer is used as ID for each vertex. The property map may contain 0 or more properties. Each vertex is connected to others by edges.

For more information see: [Edge](#), and [Tinkerpop: Property Graph Model](#)⁴⁸.

⁴⁵https://en.wikipedia.org/wiki/Stratified_sampling

⁴⁶http://en.wikipedia.org/wiki/Transaction_processing

⁴⁷http://en.wikipedia.org/wiki/Undirected_graph#Undirected_graph

⁴⁸<https://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model>

Vertex Degree From Wikipedia: [Vertex Degree](#)⁴⁹:

In graph theory, the degree (or valency) of a vertex of a graph is the number of edges incident to the vertex, with loops counted twice⁵⁰. The degree of a vertex v is denoted $\deg(v)$. The maximum degree of a graph G , denoted by $\Delta(G)$, and the minimum degree of a graph, denoted by $\delta(G)$, are the maximum and minimum degree of its vertices.

Vertex Degree Distribution From Wikipedia: [Degree Distribution](#)⁵¹:

In the study of graphs and networks, the degree of a node in a network is the number of connections it has to other nodes and the degree distribution is the probability distribution of these degrees over the whole network.

Vertices Plural form of [Vertex](#).

⁴⁹http://en.wikipedia.org/wiki/Vertex_degree

⁵⁰ Diestel, Reinhard (2005). Graph Theory (3rd ed.). Berlin, New York: Springer-Verlag. ISBN 978-3-540-26183-4.

⁵¹http://en.wikipedia.org/wiki/Degree_distribution

LEGAL STATEMENT

Copyright (c) 2015 Intel Corporation

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and limitations under the License.

CHAPTER
THIRTYTHREE

INDEX

34.1 Appendix A — Sample Application Configuration File

```
# BEGIN REQUIRED SETTINGS

trustedanalytics.atk {
#bind address - change to 0.0.0.0 to listen on all interfaces
//api.host = "127.0.0.1"

#bind port
//api.port = 9099

# The host name for the Postgresql database in which the metadata will be stored
metastore.connection-postgresql.host = "localhost"
metastore.connection-postgresql.port = "5432"
metastore.connection-postgresql.database = "ta_metastore"
metastore.connection-postgresql.username = "atkuser"
metastore.connection-postgresql.password = "MyPassword"
metastore.connection-postgresql.url =
    "jdbc:postgresql://"${trustedanalytics.atk.metastore.connection-postgresql.host}":
    "${trustedanalytics.atk.metastore.connection-postgresql.port}"/
    "${trustedanalytics.atk.metastore.connection-postgresql.database}"

# This allows for the use of postgres for a metastore.
# Service restarts will not affect the data stored in postgres.
metastore.connection = ${trustedanalytics.atk.metastore.connection-postgresql}

# This allows the use of an in memory data store.
# Restarting the REST server will create a fresh database and any
# data in the h2 DB will be lost
//metastore.connection = ${trustedanalytics.atk.metastore.connection-h2}

engine {

    # The hdfs URL where the trustedanalytics folder will be created
    # and which will be used as the starting point for any relative URLs
    fs.root = "hdfs://master.silvern.gao.cluster:8020/user/atkuser"

    # The (comma separated, no spaces) Zookeeper hosts that
    # Comma separated list of host names with zookeeper role assigned
    titan.load.storage.hostname = "node01, node02, node01"

    # Titan storage backend.
    # Available options are hbase and cassandra.
```

```
# The default is hbase.
//titan.load.storage.backend = "hbase"

# Titan storage port, defaults to 2181 for HBase ZooKeeper.
# Use 9160 for Cassandra.
titan.load.storage.port = "2181"

# The URL for connecting to the Spark master server
#spark.master = "spark://master.silvern.gao.cluster:7077"
yarn-client = "spark://master.silvern.gao.cluster:7077"

spark.conf.properties {
  # Memory should be same or lower than what is listed as available
  # in Cloudera Manager.
  # Values should generally be in gigabytes, e.g. "64g".
  spark.executor.memory = "103079215104"
}
}

# END REQUIRED SETTINGS

# The settings below are all optional.
# Some may need to be configured depending on the
# specifics of your cluster and workload.

trustedanalytics.atk {
  engine {
    auto-partitioner {
      # auto-partitioning spark based on the file size
      file-size-to-partition-size = [{ upper-bound="1MB", partitions = 15 },
                                     { upper-bound="1GB", partitions = 45 },
                                     { upper-bound="5GB", partitions = 100 },
                                     { upper-bound="10GB", partitions = 200 },
                                     { upper-bound="15GB", partitions = 375 },
                                     { upper-bound="25GB", partitions = 500 },
                                     { upper-bound="50GB", partitions = 750 },
                                     { upper-bound="100GB", partitions = 1000 },
                                     { upper-bound="200GB", partitions = 1500 },
                                     { upper-bound="300GB", partitions = 2000 },
                                     { upper-bound="400GB", partitions = 2500 },
                                     { upper-bound="600GB", partitions = 3750 }]
      # max-partitions is used if value is above the max upper-bound
      max-partitions = 10000
    }
  }

  # Configuration for the Trusted Analytics ATK REST API server
  api {
    # this is reported by the API server in the /info results -
    # it can be used to identify a particular server or cluster.
    //identifier = "ta"

    #The default page size for result pagination
    //default-count = 20

    #Timeout for waiting for results from the engine
```

```

//default-timeout = 30s

#HTTP request timeout for the REST server
//request-timeout = 29s
}

#Configuration for the processing engine
engine {
    //default-timeout = 30s
    //page-size = 1000

spark {

    # When master is empty the system defaults to spark://`hostname`:7070
    # where hostname is calculated from the current system.
    # For local mode (useful only for development testing) set master = "local[4]"
    # in cluster mode, set master and home like the example
    # master = "spark://MASTER_HOSTNAME:7077"
    # home = "/opt/cloudera/parcels/CDH/lib/spark"

    # When home is empty the system will check expected locations on the
    # local system and use the first one it finds.
    # If spark is running in yarn-cluster mode (spark.master = "yarn-cluster"),
    # spark.home needs to be set to the spark directory on CDH cluster
    # ("/usr/lib/spark", "/opt/cloudera/parcels/CDH/lib/spark/", etc)
    //home = ""

    conf {
        properties {
            # These key/value pairs will be parsed dynamically and provided
            # to SparkConf().
            # See Spark docs for possible values
            # http://spark.apache.org/docs/0.9.0/configuration.html.
            # All values should be convertible to Strings.

            #Examples of other useful properties to edit for performance tuning:

            # Increased Akka frame size from default of 10MB to 100MB to
            # allow tasks to send large results to Spark driver
            # (e.g., using collect() on large datasets).
            //spark.akka.frameSize=100

            #spark.akka.retry.wait=30000
            #spark.akka.timeout=200
            #spark.akka.timeout=30000

            //spark.shuffle consolidateFiles=true

            # Enabling RDD compression to save space (might increase CPU cycles)
            # Snappy compression is more efficient
            //spark.rdd.compress=true
            //spark.io.compression.codec=org.apache.spark.io.SnappyCompressionCodec

            #spark.storage.blockManagerHeartBeatMs=300000
            #spark.storage.blockManagerSlaveTimeoutMs=300000

            #spark.worker.timeout=600
            #spark.worker.timeout=30000

```

```
    spark.eventLog.enabled=true
    spark.eventLog.dir=
    "hdfs://master.silvern.gao.cluster:8020/user/spark/applicationHistory"
  }
}

giraph {
  #Overrides of normal Hadoop settings that are used when running Giraph jobs
  giraph.maxWorkers = 30
  //giraph.minWorkers = 1
  //giraph.SplitMasterWorker = true
  mapreduce.map.memory.mb = 4096
  mapreduce.map.java.opts = "-Xmx3072m"
  //giraph.zkIsExternal = false
}

titan {
  load {
    # documentation for these settings is available on Titan website
    # http://s3.thinkaurelius.com/docs/titan/current/titan-config-ref.html
    storage {

      # Whether to enable batch loading into the storage backend.
      # Set to true for bulk loads.
      //batch-loading = true

      # Size of the batch in which mutations are persisted.
      //buffer-size = 2048

      lock {
        # Number of milliseconds the system waits for a lock application
        # to be acknowledged by the storage backend.
        //wait-time = 400

        # Number of times the system attempts to acquire a lock before
        # giving up and throwing an exception.
        //retries = 15
      }

      hbase {
        # Pre-split settings for large datasets
        //region-count = 12
        //compression-algorithm = "SNAPPY"
      }

      cassandra {
        # Cassandra configuration options
      }
    }
  }

  ids {
    # Globally reserve graph element IDs in chunks of this size.
    # Setting this too low will make commits
    # frequently block on slow reservation requests.
    # Setting it too high will result in IDs wasted when a graph
```

```

# instance shuts down with reserved but mostly-unused blocks.
//block-size = 300000

# Number of partition block to allocate for placement of vertices.
//num-partitions = 10

# The number of milliseconds that the Titan id pool manager will
# wait before giving up on allocating a new block of ids.
//renew-timeout = 150000

# When true, vertices and edges are assigned IDs immediately upon
# creation.
# When false, IDs are assigned only when the transaction commits.
# Must be disabled for graph partitioning to work.
//flush = true

authority {
  # This setting helps separate Titan instances sharing a single
  # graph storage backend avoid contention when reserving ID
  # blocks, increasing overall throughput.
  # The options available are:
  # NONE = Default in Titan
  # LOCAL_MANUAL = Expert feature: user manually assigns each
  # Titan instance a unique conflict avoidance tag in its local
  # graph configuration.
  # GLOBAL_MANUAL = User assigns a tag to each Titan instance.
  # The tags should be globally unique for optimal performance,
  # but duplicates will not compromise correctness
  # GLOBAL_AUTO = Titan randomly selects a tag from the space of
  # all possible tags when performing allocations.
  //conflict-avoidance-mode = "GLOBAL_AUTO"

  # The number of milliseconds the system waits for an ID block
  # reservation to be acknowledged by the storage backend.
  //wait-time = 300

  # Number of times the system attempts ID block reservations
  # with random conflict avoidance tags
  # before giving up and throwing an exception
  //randomized-conflict-avoidance-retries = 10
}

auto-partitioner {
  hbase {
    # Number of regions per regionserver to set when creating
    # Titan/HBase table.
    regions-per-server = 2

    # Number of input splits for Titan reader is based on number of
    # available cores and minimum split size as follows: Number of
    # splits = Minimum(input-splits-per-spark-core * spark-cores,
    # graph size in HBase/minimum-input-splits-size-mb).
    input-splits-per-spark-core = 20
  }

  enable = true
}

```

```
}

query {
  storage {
    # query does use the batch load settings in titan.load
    backend = ${trustedanalytics.atk.engine.titan.load.storage.backend}
    hostname = ${trustedanalytics.atk.engine.titan.load.storage.hostname}
    port = ${trustedanalytics.atk.engine.titan.load.storage.port}
  }
  cache {
    # Adjust cache size parameters if you experience OutOfMemory
    # errors during Titan queries.
    # Either increase heap allocation for TrustedAnalytics Engine, or
    # reduce db-cache-size.
    # Reducing db-cache will result in cache misses and increased
    # reads from disk.
    //db-cache = true
    //db-cache-clean-wait = 20
    //db-cache-time = 180000
    #Allocates 30% of available heap to Titan (default is 50%)
    //db-cache-size = 0.3
  }
}
}
```


ERRATA

- Frame column name can accept unicode characters, but it should be avoided because some functions such as `delete_column` will fail.
- Renaming a graph to a name containing one or more of the special characters `@#$%^&*` will cause the application to hang for a long time and then raise an error.
- Attempting to create a frame with a parenthesis in the name will raise the error:

```
trustedanalytics.rest.command.CommandServerError: Job aborted due to  
stage failure: Task 7.0:5 failed 4 times, most recent failure:  
Exception failure in TID 426 on host node03.zonda.cluster:  
java.lang.IllegalArgumentException: No enum constant parquet  
.schema.OriginalType.
```

- Creating a table with an invalid source data file name causes the server to return an error message and abort, but also creates the empty (named) frame.
- When importing CSV data files to frames, small datasets may affect the number of lines skipped.

BIBLIOGRAPHY

- [R1] G. Palla, I. Derenyi, I. Farkas, and T. Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435:814, 2005 (See <http://hal.elte.hu/cfinder/wiki/papers/communitylettm.pdf>)
- [R2] Varamesh, A.; Akbari, M.K.; Fereiduni, M.; Sharifian, S.; Bagheri, A., “Distributed Clique Percolation based community detection on social networks using MapReduce,” *Information and Knowledge Technology (IKT)*, 2013 5th Conference on, vol., no., pp.478,483, 28-30 May 2013

Symbols

__init__() (built-in function), 173, 209, 211, 247, 288, 291, 304, 314, 317, 321, 324, 327, 335, 338, 341, 344, 355, 358, 362, 364, 366, 367, 369
 __init__() (trustedanalytics.CsvFile method), 161
 __init__() (trustedanalytics.HBaseTable method), 164
 __init__() (trustedanalytics.HiveQuery method), 163
 __init__() (trustedanalytics.JdbcTable method), 165
 __init__() (trustedanalytics.JsonFile method), 166
 __init__() (trustedanalytics.LineFile method), 167
 __init__() (trustedanalytics.Pandas method), 168
 __init__() (trustedanalytics.XmlFile method), 170
 __private_ml, 291, 304
 __private_query, 305

A

add column, 28
 example, 28
 add_columns() (built-in function), 174, 211, 248
 add_edges() (built-in function), 176
 add_vertices() (built-in function), 213
 Adjacency List, **633**
 administration, 77
 Aggregation Function, **633**
 Alpha, **633**
 Alternating Least Squares, **633**
 Anaconda, 143
 analytics
 graph, 34
 annotate_degrees() (built-in function), 291, 305
 annotate_weighted_degrees() (built-in function), 292, 306
 API Maturity Tags, **633**
 append, 24
 example, 24
 append() (built-in function), 249
 ASCII, **634**
 assign_sample() (built-in function), 176, 213, 250
 Average Path Length, **634**

B

Bayesian Inference, **634**

Belief Propagation, **634**
 benchmarking, 75, 141
 Beta, **634**
 Bias vs Variance, **634**
 Bias-variance tradeoff, **634**
 bin_column() (built-in function), 177, 214, 251
 bin_column_equal_depth() (built-in function), 178, 215, 252
 bin_column_equal_width() (built-in function), 179, 215, 253
 binary logistic regression, 40

C

categorical_summary() (built-in function), 179, 216, 253
 Central Tendency, **634**
 Centrality, **634**
 Centrality (Katz), **634**
 Centrality (PageRank), **634**
 Character-Separated Values, **634**
 characters
 special, 655
 Classification, **634**
 classification, 40
 classification_metrics() (built-in function), 180, 217, 255
 cloudera manager, 84
 cluster, 85, 87, 96
 cluster, configuration, 62, 128
 Clustering, **634**
 clustering_coefficient() (built-in function), 293, 307
 Collaborative Clustering, **634**
 Collaborative Filtering, **634**
 CollaborativeFilteringModel (built-in class), 332
 column names, 655
 column_median() (built-in function), 182, 219, 256
 column_mode() (built-in function), 182, 219, 256
 column_names, 183, 220, 257
 column_summary_statistics() (built-in function), 183, 220, 258
 Comma-Separated Variables, **635**
 Community Structure Detection, **635**
 compute_misplaced_score() (built-in function), 185, 222, 259

Confusion Matrices, [635](#)
Confusion Matrix, [635](#)
Conjugate Gradient Descent, [635](#)
connect, [155](#)
connect() (in module trustedanalytics), [155](#)
Connected Component, [635](#)
Convergence, [635](#)
copy() (built-in function), [185](#), [222](#), [259](#), [294](#), [308](#)
correlation() (built-in function), [186](#), [223](#), [260](#)
correlation_matrix() (built-in function), [187](#), [223](#), [261](#)
count() (built-in function), [187](#), [224](#), [261](#)
covariance() (built-in function), [187](#), [224](#), [261](#)
covariance_matrix() (built-in function), [188](#), [224](#), [262](#)
CSV, [22](#), [635](#)
CsvFile (class in trustedanalytics), [161](#)
cumulative_percent() (built-in function), [188](#), [225](#), [262](#)
cumulative_sum() (built-in function), [188](#), [225](#), [262](#)

D

data
 type, [22](#), [157](#)
database, [89](#)
define_edge_type() (built-in function), [295](#)
define_vertex_type() (built-in function), [295](#)
Degree, [635](#)
Deprecated, [635](#)
develop, [53](#)
Directed Acyclic Graph (DAG), [635](#)
dot_product() (built-in function), [189](#), [225](#), [263](#)
download() (built-in function), [189](#), [226](#), [263](#)
drop column, [28](#)
 example, [28](#)
drop duplicates, [27](#)
 example, [27](#)
drop rows, [27](#)
 example, [27](#)
drop_columns() (built-in function), [190](#), [227](#), [264](#)
drop_duplicates() (built-in function), [190](#), [227](#), [264](#)
drop_frames() (built-in function), [289](#)
drop_graphs() (built-in function), [315](#)
drop_models() (built-in function), [370](#)
drop_rows() (built-in function), [191](#), [227](#), [265](#)
drop_vertices() (built-in function), [228](#)
duplicates, [27](#)

E

ECDF, [635](#)
ecdf() (built-in function), [191](#), [228](#), [265](#)
Eclipse, [122](#), [145](#), [147](#)
Edge, [635](#)
edge_count, [295](#)
EdgeFrame (built-in class), [208](#)
edges, [296](#)
Empirical Cumulative Distribution, [635](#)

enhance, [53](#)
entropy() (built-in function), [191](#), [229](#), [265](#)
Enumerate, [635](#)
Equal Depth Binning, [635](#)
Equal Width Binning, [636](#)
example, [20](#)
 add column, [28](#)
 append, [24](#)
 drop column, [28](#)
 drop duplicates, [27](#)
 drop rows, [27](#)
 filter rows, [27](#)
 flatten column, [32](#)
 Frame (capital F), [23](#)
 frame (lower case f), [23](#)
 group by, [29](#)
 join, [30](#)
 rename column, [28](#)
export_to_csv() (built-in function), [192](#), [229](#), [266](#)
export_to_graph() (built-in function), [308](#)
export_to_hbase() (built-in function), [192](#), [229](#), [266](#)
export_to_hive() (built-in function), [193](#), [230](#), [267](#)
export_to_jdbc() (built-in function), [193](#), [230](#), [267](#)
export_to_json() (built-in function), [193](#), [231](#), [267](#)
export_to_titan() (built-in function), [296](#)
extend, [53](#)
extending, development, [43](#)
Extract, Transform, and Load, [636](#)

F

F-Measure, [636](#)
F-Score, [636](#)
F1 Score, [636](#)
field_names (trustedanalytics.CsvFile attribute), [162](#)
field_names (trustedanalytics.Pandas attribute), [169](#)
field_types (trustedanalytics.CsvFile attribute), [163](#)
field_types (trustedanalytics.Pandas attribute), [169](#)
filter rows, [27](#)
 example, [27](#)
filter() (built-in function), [194](#), [231](#), [268](#)
flatten column, [32](#)
 example, [32](#)
flatten_column() (built-in function), [194](#), [231](#), [268](#)
float32, [636](#)
float64, [636](#)
Frame (built-in class), [287](#)
Frame (capital F), [23](#), [636](#)
 example, [23](#)
frame (lower case f), [23](#), [636](#)
 example, [23](#)

G

GaBP, [636](#)
Gaussian Belief Propagation, [636](#)

Gaussian Distribution, [636](#)
 Gaussian Random Fields, [636](#)
[get_error_frame\(\)](#) (built-in function), [195](#), [232](#), [269](#)
[get_frame\(\)](#) (built-in function), [289](#)
[get_frame_names\(\)](#) (built-in function), [290](#)
[get_graph\(\)](#) (built-in function), [314](#)
[get_graph_names\(\)](#) (built-in function), [315](#)
[get_model\(\)](#) (built-in function), [369](#)
[get_model_names\(\)](#) (built-in function), [370](#)
 Giraph, [75](#), [141](#)
 Global Clustering Coefficient, [636](#)
 Graph, [637](#)
 graph
 analytics, [34](#)
 Graph (built-in class), [303](#)
 Graph Analytics, [637](#)
 Graph Database Directions, [637](#)
 Graph Element, [637](#)
[graph_clustering\(\)](#) (built-in function), [308](#)
[graphx_connected_components\(\)](#) (built-in function), [296](#), [308](#)
[graphx_pagerank\(\)](#) (built-in function), [297](#), [309](#)
[graphx_triangle_count\(\)](#) (built-in function), [299](#), [311](#)
 Gremlin, [637](#)
 group by, [29](#)
 example, [29](#)
[group_by\(\)](#) (built-in function), [195](#), [232](#), [269](#)

H

H2, [47](#), [103](#)
 HBase, [76](#), [142](#), [637](#)
 HBaseTable (class in trustedanalytics), [164](#)
[histogram\(\)](#) (built-in function), [197](#), [234](#), [271](#)
 HiveQuery (class in trustedanalytics), [163](#)
 host, [85](#), [87](#)

I

IDE, [145](#), [147](#), [149](#), [151](#)
 importing, [655](#)
[inspect\(\)](#) (built-in function), [198](#), [235](#), [272](#)
 installation, [94](#)
 int32, [637](#)
 int64, [637](#)
 IntelliJ, [145](#), [149](#)
 IPython, [49](#)
 Ising Smoothing Parameter, [638](#)

J

Java, [85](#)
 JdbcTable (class in trustedanalytics), [165](#)
 join, [30](#)
 example, [30](#)
[join\(\)](#) (built-in function), [199](#), [236](#), [273](#)
 JSON, [22](#), [638](#)

JsonFile (class in trustedanalytics), [166](#)

K

K-S Test, [638](#)
 Katz Centrality, [638](#)
 KMeansModel (built-in class), [340](#)

L

Label Propagation, [638](#)
[label_propagation\(\)](#) (built-in function), [274](#)
 Labeled Data vs Unlabeled Data, [638](#)
 Lambda, [638](#)
 Lambda Function, [639](#)
 Latent Dirichlet Allocation, [639](#)
 LdaModel (built-in class), [351](#)
 Least Squares, [639](#)
 LibsvmModel (built-in class), [320](#)
 LinearRegressionModel (built-in class), [366](#)
 LineFile, [22](#), [639](#)
 LineFile (class in trustedanalytics), [167](#)
[loadhbase\(\)](#) (built-in function), [200](#), [237](#), [276](#)
[loadhive\(\)](#) (built-in function), [201](#), [238](#), [277](#)
[loadjdbc\(\)](#) (built-in function), [201](#), [238](#), [277](#)
 Local Clustering Coefficient, [639](#)
 log, [123](#)
 LogisticRegressionModel (built-in class), [362](#)
 Loopy Belief Propagation, [639](#)
[loopy_belief_propagation\(\)](#) (built-in function), [277](#)

M

machine
 virtual, [111](#)
 Machine Learning, [639](#)
 machine learning, [37](#), [39](#)
 MapReduce, [639](#)
 Markov Random Fields, [640](#)
 metadata, [47](#)
[ml.belief_propagation\(\)](#) (built-in function), [299](#), [311](#)
[ml.kclique_percolation\(\)](#) (built-in function), [300](#)
 model, [39](#)

N

NaiveBayesModel (built-in class), [364](#)
 name, [202](#), [239](#), [281](#), [302](#), [312](#), [317](#), [321](#), [325](#), [330](#), [338](#), [341](#), [348](#), [359](#), [363](#), [365](#), [367](#)
 Normal Distribution, [636](#)

O

OLAP, [640](#)
 OLTP, [640](#)

P

PageRank, [640](#)

PageRank Centrality, [640](#)
Pandas (class in trustedanalytics), [168](#)
parcel, [84](#), [85](#)
performance, [75](#), [76](#), [141](#), [142](#)
plugin, [53](#)
PostgreSQL, [47](#), [49](#), [103](#)
Precision/Recall, [640](#)
predict() (built-in function), [318](#), [322](#), [325](#), [339](#), [342](#),
 [349](#), [359](#), [363](#), [365](#), [367](#)
prediction, [40](#)
PrincipalComponentsModel (built-in class), [326](#)
Property Map, [640](#)
proxy, [82](#), [84](#)
publish() (built-in function), [318](#), [322](#), [326](#), [339](#), [349](#), [368](#)
PyCharm, [145](#), [151](#)
PyDev, [122](#)
Python, [21](#), [35](#), [41](#), [143](#), [152](#)
 Windows, [143](#)
Python User-defined Function, [640](#)

Q

Quantile, [640](#)
quantiles() (built-in function), [202](#), [239](#), [281](#)
query.gremlin() (built-in function), [312](#)

R

RandomForestClassifierModel (built-in class), [324](#)
RandomForestRegressorModel (built-in class), [369](#)
RDF, [640](#)
Receiver Operating Characteristic, [640](#)
recommend() (built-in function), [331](#)
Recommendation Systems, [641](#)
rename column, [28](#)
 example, [28](#)
rename_columns() (built-in function), [202](#), [239](#), [281](#)
repository, [85](#), [96](#), [97](#)
Resource Description Framework, [641](#)
REST, [370](#), [617](#)
REST server, [105](#)
ROC, [641](#)
role assignment, [89](#)
Row Functions, [641](#)
row_count, [203](#), [240](#), [282](#)

S

Schema, [641](#)
schema, [203](#), [240](#), [282](#)
score() (built-in function), [318](#)
semi-supervised, [40](#)
Semi-Supervised Learning, [641](#)
services, [87](#)
Simple Random Sampling, [641](#)
Smoothing, [641](#)
sort() (built-in function), [203](#), [240](#), [282](#)

sorted_k() (built-in function), [204](#), [241](#), [283](#)
special
 characters, [655](#)
statistics, [29](#)
status, [205](#), [242](#), [284](#), [302](#), [313](#)
Stratified Sampling, [642](#)
Superstep, [642](#)
Supersteps, [642](#)
supervised, [40](#)
Supervised Learning, [642](#)
SvmModel (built-in class), [343](#)

T

Tab-Separated Variables, [642](#)
take() (built-in function), [206](#), [243](#), [285](#)
tally() (built-in function), [206](#), [243](#), [285](#)
tally_percent() (built-in function), [207](#), [244](#), [286](#)
test() (built-in function), [318](#), [322](#), [342](#), [360](#)
TitanGraph, [642](#)
TitanGraph (built-in class), [314](#)
top_k() (built-in function), [207](#), [244](#), [286](#)
Topic Modeling, [642](#)
train() (built-in function), [319](#), [323](#), [326](#), [331](#), [340](#), [343](#),
 [350](#), [360](#), [363](#), [365](#), [368](#)
Transaction Processing, [642](#)
Transactional Functionality, [642](#)
tuning, [75](#), [76](#), [141](#), [142](#)
type
 data, [22](#), [157](#)

U

UDF, [35](#), [642](#)
Undirected Graph, [642](#)
unflatten_column() (built-in function), [208](#), [245](#), [287](#)
Unicode, [642](#)
unsupervised, [40](#)
Unsupervised Learning, [642](#)

V

Vertex, [642](#)
Vertex Degree, [643](#)
Vertex Degree Distribution, [643](#)
vertex_count, [302](#)
vertex_sample() (built-in function), [313](#)
VertexFrame (built-in class), [245](#)
Vertices, [643](#)
vertices, [303](#)
virtual
 machine, [111](#)

W

Windows
 Python, [143](#)

X

XmlFile (class in trustedanalytics), [170](#)