**Welcome to the Intel® Data Platform: Analytics Toolkit**

The Intel® Data Platform: Analytics Toolkit provides a unified environment for analyzing "Big Data" that brings together the powerful new capabilities of graph analytics with classic machine learning. This new toolkit enables data scientists and developers to more easily create new analytic capabilities in a platform that is more cost effective and easier to manage than disparate, non-integrated tools. Users can leverage the full spectrum of powerful analytic capabilities without constraining their sophistication and creativity.

Contents

[**Welcome to the Intel® Data Platform: Analytics Toolkit** 1](#_Toc381716053)

[Getting Started 3](#_Toc381716054)

[Features 3](#_Toc381716055)

[Built-in Databases 3](#_Toc381716056)

[Before You Start 3](#_Toc381716057)

[Graph Database 5](#_Toc381716058)

[Titan 5](#_Toc381716059)

[Graph Analytics 5](#_Toc381716060)

[Gremlin 5](#_Toc381716061)

[Data Flow 6](#_Toc381716062)

[Data Import 6](#_Toc381716063)

[Feature Engineering 6](#_Toc381716064)

[Graph Construction 8](#_Toc381716065)

[Machine Learning 10](#_Toc381716066)

[Machine Learning as the Analytics Toolkit Uses It 10](#_Toc381716067)

[Machine Learning Algorithms 13](#_Toc381716068)

[Collaborative Filtering 13](#_Toc381716069)

[Graph Analytics 24](#_Toc381716070)

[Graphical Models 27](#_Toc381716071)

[Topic Modeling 34](#_Toc381716072)

[Perform Analytics on the Graph 40](#_Toc381716073)

[Glossary 47](#_Toc381716074)

Within the toolkit, Intel Graph Analytics provides an industry-first environment for creating and analyzing big data graphs across the data science workflow.

You will import, then clean and feature engineer the data (the actual data crunching), and finally produce graphs and run analytics on those graphs.

The Analytics Toolkit is built on open source components for ease of use and integration with whatever tools you might be using now. We use:

* Apache™ Hadoop® and the Hadoop Distributed File System (HDFS).
* Apache™ Giraph® scalable iterative graph processing system.
* A Python™-based API, using iPython.
* Aurelius™ Titan® Graph Database.
* TinkerPop™ Gremlin® Graph Query language.

These tools work well together, and with many other open source tools.

**Note**: For the [GraphTrial.Intel.com](https://graphtrial.intel.com/) release, no installation is necessary.

# Getting Started

The Intel® Data Platform: Analytics Toolkit is a scalable database and analytics tool for examining and analyzing "Big Data." The Analytics Toolkit enables users to build graphs, examine the relationships between data, and view the data in different ways. The Intel® Data Platform: Analytics Toolkit includes powerful algorithms for transforming your data and a built-in graph database allows you to examine your data and run analytic tools on it.

## Features

We add features to the Analytics Toolkit frequently, so check back for updates.

* You can import data from several different formats.
* The Analytics Toolkit provides data cleaning tools to prepare your data by removing erroneous values, transforming value to a normalized state and constructing new features through manipulating existing values.
* We provide powerful algorithms for gaining deeper insight into your data.

## Built-in Databases

The Intel® Data Platform: Analytics Toolkit holds data as either traditional table-based format in HBase or can store that data in the Titan graph database. The Intel® Data Platform: Analytics Toolkit provides the functionality that allows data scientists to easily manipulate data in either of these data formats.

## Before You Start

You should be familiar with some of the Python packages already available in the open source community.

The Intel® Data Platform: Analytics Toolkit provides a set of functionality exposed through a Python API. You can access this API through iPython notebooks, so familiarizing yourself with iPython will be helpful to you. <http://ipython.org/>

You’ll want to know about Hadoop and its various components. We use HDFS, MapReduce, and YARN.  
<http://hadoop.apache.org/docs/current/index.html>

We use Apache Giraph for graph-based machine learning.  
<http://giraph.apache.org/>

We use the Titan Graph Database from Think Aurelius because it is flexible and highly scalable.  
<http://thinkaurelius.github.io/titan/>

When using Titan, you can access the database with the Gremlin graph query language from TinkerPop. You can write Gremlin queries to investigate and manipulate your graphs.  
<https://github.com/tinkerpop/gremlin/wiki>

The toolkit notebooks run Python 2.7 and come with several python packages installed, ready to import in your interactive sessions:

|  |  |  |
| --- | --- | --- |
| **Python Package** | **Description** | **URL** |
| scipy | scientific computing | <http://www.scipy.org> |
| numpy | numeric computing | <http://www.numpy.org/> |
| sympy | symbolic math | <http://www.sympy.org> |
| pandas | data structures | <http://pandas.pydata.org/> |
| matplotlib | plotting | <http://matplotlib.org> |
| nltk | natural language toolkit | <http://www.nltk.org/> |
| jinja2 | templating engine | <http://jinja.pocoo.org/> |
| bulbs\* | graph database support | <http://bulbflow.com/docs/> |
| happybase | HBase support | <http://happybase.readthedocs.org/en/latest/> |
| pydoop | Hadoop support | <http://pydoop.sourceforge.net/docs/> |
| mrjob | Map Reduce | <https://pythonhosted.org/mrjob/> |

# Graph Database

The Intel® Data Platform: Analytics Toolkit uses a graph database to handle complex, non-tabular data. Graph databases are described as a series of vertices connected by edges. We use these terms throughout our documentation. Each vertex is a data element, like a field in a relational database. The edges connect the vertices and show the relationships between them. You might use this to perform organizational analysis, or to analyze business, or political relationships.

## Titan

The Intel® Data Platform: Analytics Toolkit uses the open source Titan Graph Database, from Aurelius. See <http://thinkaurelius.github.io/titan/> for more details.

The Titan graph database does not use indices and thus, does not use index lookups. Each data element (vertex) has a pointer (edge) to the element adjacent to it. This is great for handling graphs, and most Big Data problems are graph data. See <http://en.wikipedia.org/wiki/Graph_data_structure>.

## Graph Analytics

**Graph analytics** are the broad category of useful calculations you use to examine a graph. Examples of graph analytics may include traversals — algorithmic walk throughs of the graph to determine optimal paths and relationship between vertices, and statistics — that determine 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 such that calculations are iterated across the nodes of the graph to uncover patterns and relationships, such as finding similarities based on relationships, or recursively optimizing some parameter across nodes.

## Gremlin

Gremlin is an open source, graph database, query language. Think about it as SQL queries for graph databases. See the official Gremlin page here: <https://github.com/tinkerpop/gremlin/wiki>. And the Titan Gremlin page here: <https://github.com/thinkaurelius/titan/wiki/Gremlin-Query-Language>.

# Data Flow

When using the Intel® Data Platform: Analytics Toolkit, you will import your data, clean the data, combine or filter the data, and finally, make a graph of the data.

## Data Import

Use the FrameBuilder component to upload data. In this release, we support CSV (Comma Separated Value), JSON, and XML formats. You will upload your data to the server through the file upload feature provided on [GraphTrial.Intel.com](https://graphtrial.intel.com/). We will store your file on the HDFS file system of your instance, after which you can pull the data using FrameBuilder. When you complete this step, your data will be accessible thorough a BigDataFrame object.

For example, here is how we import a CSV file:

    from intel\_analytics.table.bigdataframe import get\_frame\_builder  
    fb = get\_frame\_builder()  
    csvfile = '/user/hadoop/movie\_recommendations\_raw.csv'  
    frame = fb.build\_from\_csv('myframe',  
                              csvfile,  
                              schema='user:long,vertex\_type:chararray,movie:long,rating:long,splits:chararray',  
                              overwrite=True)

To start, we import the frame builder, that is, the object that transforms your data set into a table. We use this fb object to create a BigDataFrame object that will store your data in an underlying HBase table.

Then we use the FrameBuilder to transform the data in the csv file to a BigDataFrame stored in an underlying HBase table. For other data types, you will use build\_from\_json for JSON files or build\_from\_xml for XML files.

In the last step of the example, 'myframe' is the table name of your data frame. The csv file has now been copied into the BigDataFrame and is ready to be cleaned and transformed using the advanced functionality of the BigDataFrame. The schema line describes the data in the csv file, so the BigDataFrame knows the type of data available. We set the overwrite flag to true so that the imported data overwrites the data already in the 'myframe' BigDataFrame, if any was present.

## Feature Engineering

Feature Engineering is an iterative process in which you gather data, clean it, run algorithms on it, and then look at the results. Then, you'll look at what you have and iterate again, looking for more data, or removing some data from the set until you have the desired result. You'll probably clean the data in a number of different ways, and then run your transforms again. We provide several methods in the Analytics Toolkit, but you can use features from other Python libraries as well to manipulate your data. The Intel® Data Platform: Analytics Toolkit Python libraries have been specifically designed to handle very large data sets, so when using standard Python libraries, be aware that some of them are not designed to handle the very large data sets stored in Hadoop.

### Data Cleaning

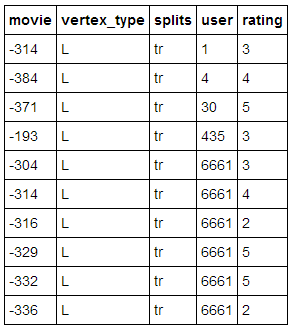
First, to clean your data, you will want to remove incomplete, incorrect, inaccurate, or corrupted data from your data set. You will use the BigDataFrame API to perform the data cleaning.

In the following examples, we entered the commands into the Intel® Data Platform: Analytics Toolkit iPython notebook.

Here's an example of cleaning data:

    frame.dropna()  
    HTML(frame.inspect\_as\_html())

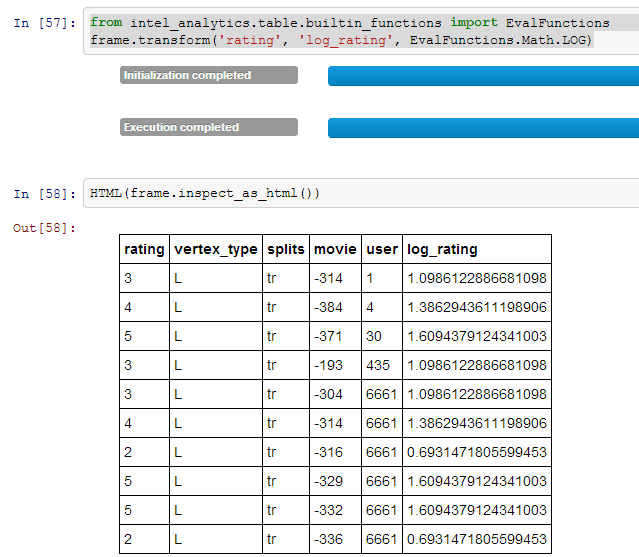
The first line performs the actual cleaning. In this case, the dropna() method removes all rows from the table that contain NA results in any column. The second line displays a sampling of the data frame's contents. In this case:



During the “cleaning phase,” you will not only want to remove extraneous or erroneous data, you will want to take existing values and transform them into features you can use. This is where you manipulate the data, that is, actually crunch the data. You will use the BigDataFrame API calls to manipulate your data.

### Feature Engineering or Data Transformation

The figure below shows part of an iPython notebook, transforming a movie rating to a logarithmic value, and displaying the results as an HTML output. The frame.transform method creates a new log\_rating column that is the result of applying a LOG transform to the original rating column.



## Graph Construction

Once you have imported your data, cleaned it, performed feature engineering on it (that is, manipulated the data), and now you are at the point where you can make a graph. You will use the BigGraph API calls to store the data in a graph.

There are two main steps to graph construction. First, you will configure graph builder to describe the graph, and then you build it.

### Configure Your Graph

You configure the graph builder that is going to create your graph. You are creating the recipe to build your graph by specifying how to manipulate the data in order to create this particular graph.

The first step specifies the sources for the vertices and edges, their labels and properties.

The code below shows an example of how to register (create) a graph from the BigDataFrame object where it contains columns that have users, movies, and ratings.

    from intel\_analytics.graph.biggraph import get\_graph\_builder, GraphTypes  
    gb = get\_graph\_builder(GraphTypes.Property, frame)  
    gb.register\_vertex('user',['vertex\_type'])  
    gb.register\_vertex('movie')  
    gb.register\_edge(('user', 'movie', 'rates'), ['splits','rating'])

### Build Your Graph

After you have described the graph that you want to create, the second step is to build the graph and load its data into graph database.

To build a graph, we need only one line of code:

    graph = gb.build("mygraph", overwrite=True)

We name the graph mygraph, and we set overwrite to True, as above, to overwrite any existing graph with this name.

# Machine Learning

Machine learning is a branch of artificial intelligence. It is about constructing and studying software that can "learn" from data.

When you enter a search phrase or question in an internet search tool, you get a list of websites ranked in order of relevance. The search engine bases its list on the content of the site, the links in each sites' webpages, how often users visit these sites, "how often users follow the suggested links in a query, and examples of queries and manually ranked webpages." More and more, machine learning is being used to automate search engines. (See <http://alex.smola.org/drafts/thebook.pdf>.)

Internet bookstores or video rental sites use collaborative filtering to encourage users to buy more books, or rent more movies. The website needs to produce a list of suggestions, without having access to a specific query. So the site needs to use the customer's past purchasing behavior to predict future behavior. Additionally, the site might use the behavior of similar users to predict what this customer might find interesting. The business goal here is to automate this logic, so that the application behind the website can make these predictions without involving an analyst or other website personnel.

There are many other problems that are amenable to machine learning solutions. Translation of text for example is a difficult issue. It is possible to use examples of pre-translated to text to enable an algorithm to "learn" how to translate text from one language to another. This requires many good examples of translations, but ultimately, the software learns how to translate, at least for specific languages.

To solve these and other problems, we need to be able to build software that can "learn" from data. We also want to consider each problem by the type of data involved. That way, when we encounter new problems, we can look at the type of data and previous solutions. Even when we have similar problems, the data may use different measures, feet, inches, meters, pounds, kilograms, Euros, yen, dollars, or pesos. To automate these problems and the solutions, we need to normalize the data.

## Machine Learning as the Analytics Toolkit Uses It

There is plenty of literature on machine learning for those who want to gain a more thorough understanding of it. We recommend: <http://alex.smola.org/drafts/thebook.pdf> and <http://en.wikipedia.org/wiki/Machine_learning>. You might find this link helpful as well:  
<http://blog.bigml.com/2013/02/21/everything-you-wanted-to-know-about-machine-learning-but-were-too-afraid-to-ask-part-two/>.

The API page for our use of machine learning is here:  
<https://www.graphtrial.intel.com/documentation/pythonapi>  
Refer to the intel\_analytics.graph.titan.ml module link.

### Algorithm Types

The algorithms that we use in the Analytics Toolkit are Unsupervised Learning or Supervised Learning algorithms, where you either have definitive results (supervised) or where the results are determined by an estimation on the relationship of the data to be examined and not a specific known result (unsupervised).

For more information <http://en.wikipedia.org/wiki/Machine_learning#Algorithm_types> has a longer discussion of algorithm types.

* **Supervised Learning** — These algorithms are designed to teach the software to generalize from specific data. Then the generalized learning is applied to new problems.
* **Unsupervised Learning** — Here the algorithm learns from data where the outcome is unknown. The idea here is to find new structure in the data.
* **Semi-Supervised Learning** — In this case, some of the data given to the algorithm are known, as in supervised learning, and some are unknown, as in unsupervised learning. The idea here is that the software learns faster.

We implement machine learning in several classes and several methods. The following methods implement various algorithms for manipulating your data.

* The recommend() method is for making recommendations, such as movies, books, or guitars the user might find interesting.
* The belief\_prop() method performs belief propagation in a loop until the algorithm converges. You can use this for performing inference on graphical models, such as Bayesian networks and Markov random fields.
* The page\_rank() method is for ranking pages, as in a websearch. You can find details on this algorithm here:  
  <http://en.wikipedia.org/wiki/PageRank>.
* The avg\_path\_len() method calculates the average path length, that is, “the average number of steps along the shortest paths for all possible pairs of network nodes.” See <http://en.wikipedia.org/wiki/Average_path_length>.
* The label\_prop() method performs label propagation on Gaussian random fields. Use this method to detect community structure in networks. For more details, see <http://reports-archive.adm.cs.cmu.edu/anon/cald/abstracts/02-107.html>.
* The lda() method performs latent Dirichlet allocation. For more information, see <http://en.wikipedia.org/wiki/Latent_Dirichlet_allocation>.
* The als() method performs the Alternating Least Squares with Bias for collaborative filtering algorithms. Use this for recommendation calculations. For more details, see:  
  [http://www.hpl.hp.com/personal/Robert\_Schreiber/papers/2008%20AAIM%20Netflix/netflix\_aaim08(submitted).pdf](http://www.hpl.hp.com/personal/Robert_Schreiber/papers/2008%20AAIM%20Netflix/netflix_aaim08%28submitted%29.pdf)  
  and [http://public.research.att.com/~volinsky/netflix/kdd08koren.pdf](http://public.research.att.com/%7Evolinsky/netflix/kdd08koren.pdf).
* The cgd() method performs conjugate gradient descent filtering. Use this for recommendation calculations. For more information see  
  [http://public.research.att.com/~volinsky/netflix/kdd08koren.pdf](http://public.research.att.com/%7Evolinsky/netflix/kdd08koren.pdf).

### Model Evaluation

Different models need different evaluation methods. What will be added in the first step will be prior and posterior histogram, RoC/AUC curves, and lift curves.

# Machine Learning Algorithms

In this release of the Intel® Data Platform: Analytics Toolkit, we support eight graphical algorithms in iGiraph. From a functionality point of view, they fall into these categories: Collaborative Filtering, Graph Analytics, Graphical Models, and Topic Modeling.

**Collaborative Filtering**

* [Alternating Least Squares](#_Alternating_Least_Squares)
* [Conjugate Gradient Descent](#_Conjugate_Gradient_Descent)

**Graph Analytics**

* [Average Path Length](#_Average_Path_Length)
* [Connected Components](#_Connected_Components)
* [Page Rank](#_Page_Rank)

**Graphical Models**

* [Label Propagation](#_Label_Propagation)
* [Loopy Belief Propagation](#Loopy_Belief_Propagation)

**Topic Modeling**

* [Latent Dirichlet Allocation](#_Latent_Dirichlet_Allocation)

## Collaborative Filtering

Collaborative filtering (CF) is widely used in recommender systems.  
See: <http://en.wikipedia.org/wiki/Collaborative_filtering>

We support two collaborative filtering algorithms: Alternating Least Square (ALS) and Conjugated Gradient Descent (CGD).

### Alternating Least Squares

We use the Alternating Least Squares with Bias for collaborative filtering algorithms.  
<http://columbiadatascience.com/2012/10/18/week-7-hunch-com-recommendation-engines-svd-alternating-least-squares-convexity-filter-bubbles/>

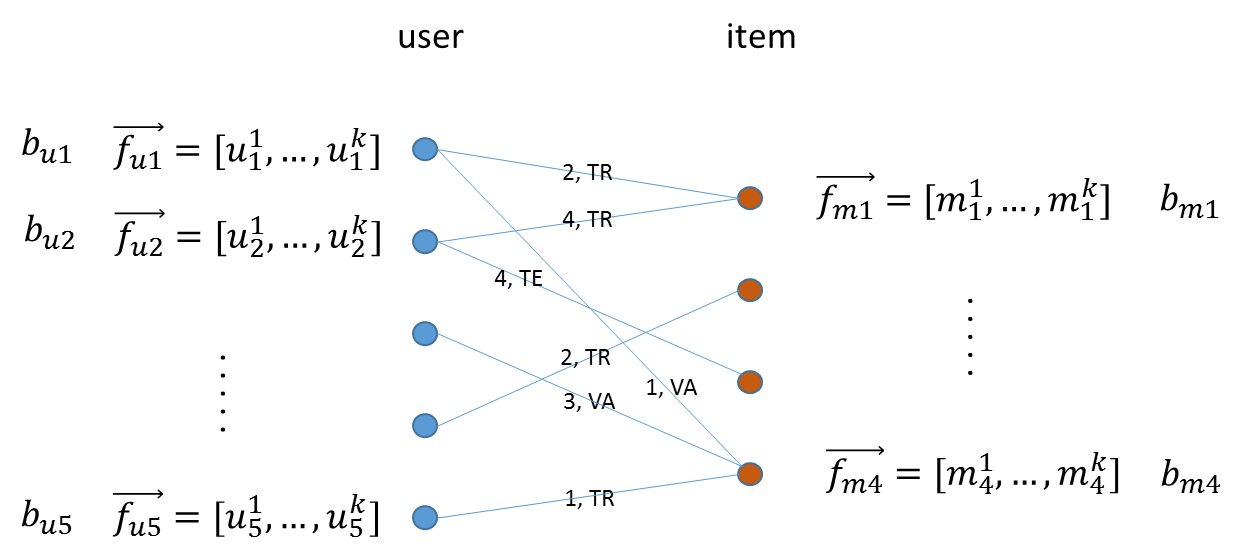
Our implementation is based on the following paper.  
Y. Zhou, D. Wilkinson, R. Schreiber and R. Pan. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. 2008.  
Y. Koren. Factorization Meets the Neighborhood: a Multifaceted Collaborative Filtering Model. In ACM KDD 2008. (Equation 5)

[http://public.research.att.com/~volinsky/netflix/kdd08koren.pdf](http://public.research.att.com/%7Evolinsky/netflix/kdd08koren.pdf)

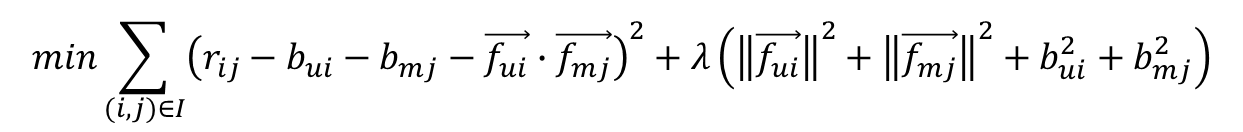
This algorithm for collaborative filtering is widely used in recommendation systems to suggest items (products, movies, articles, and so on) to potential users based on historical records of items that all users have purchased, rated, or viewed. The records are usually organized as a preference matrix P, which is a sparse matrix holding the preferences (such as, ratings) given by users to items. Within collaborative filtering approaches, ALS falls in the category of the 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.

### The ALS Model

A typical representation of the preference matrix P in Giraph is a bipartite graph, where nodes at the left side represent a list of users and nodes at the right side represent a set of items (such as, 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”.

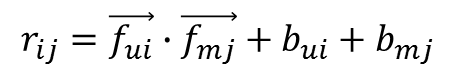


After executing ALS on the input bipartite graph, each node in the graph will be associated with a vector of length k, where k is the feature dimension is specified by the user, and a bias term . ALS optimizes and alternatively between user profiles and item profiles such that the following l2 regularized cost function is minimized:



Here the first term strives to find ’s and ’s that fit the given ratings, and the second term (l2 regularization) 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 and are determined, given an item mj the rating from user ui can be predicted by a simple linear model:



### ALS Example Usage

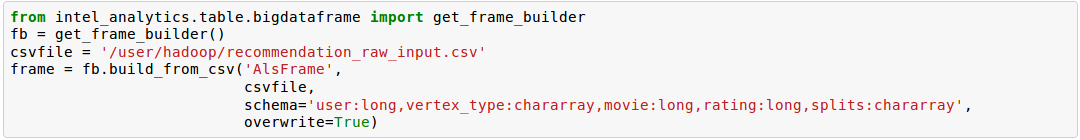
#### Input data format

The ALS algorithm takes an input data represented in CSV, JSON or XML format. We use aCSV file as an example. Each CSV file consists of at least five columns as shown in the example below. The user column is a list of user IDs. The movie column is a list of movie IDs. The rating column records how the user rates the movie in each row. The vertex\_type labels the type of the source vertex in each row. It labels which nodes will be on the "left-side" and which nodes will be on the "right-side" in the bi-partite graph we are building. The splits column specifies this row of data is for train, validation, or test. We used TR, VA, TE for these three types of splits, respectively.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **user** | **movie** | **rating** | **vertext\_type** | **splits** |
| 1 | -314 | 3 | L | TR |
| 4 | -384 | 4 | L | TR |
| 30 | -371 | 5 | L | TR |
| 435 | -193 | 3 | L | TR |
| 6661 | -304 | 3 | L | TR |
| 6661 | -314 | 4 | L | TR |
| 6661 | -316 | 2 | L | TR |
| 6661 | -329 | 5 | L | TR |
| 6661 | -332 | 5 | L | TR |

#### Data import

To import the ALS input data, use the following ipython calls we provided.

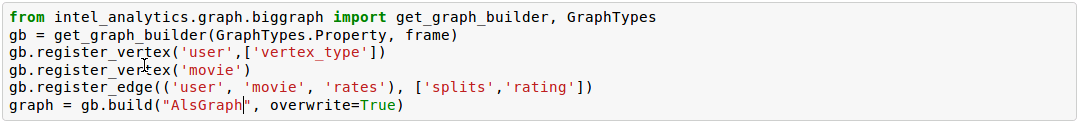


The example above loads the ALS input data from a CSV file. The first line imports the needed python modules. The second line gets the frame builder into the fb object. The third line specifies the path to the input file. The rest of the lines import the input data. Here is a detailed description of the build\_from\_csv method.

* The first argument is the name you want to give to the frame. We used AlsFrame in this example.
* The second argument specifies that this is a csv file.
* The third argument is the schema of the input data. You need to name each column, and specify the data type of each column in your input CSV input data.
* The fourth argument is whether to overwrite the frame if you have imported data to the AlsFrame before.

#### Graph Construction

After you import the raw data, you register which fields to use for source vertex, which fields to use for target vertex, and then construct a graph from your input data.



In the example above, the first two lines import python modules related to graph construction, and get the graph builder object into gb. The third to fifth lines register the graph. Line three registers user column as the source vertex and registers the vertex property vertex\_type to this vertex. Line four registers movie column as the target vertex. The fifth line registers an edge from user to movie, with the label rates. Additionally, rating and splits are two edge properties registered for this algorithm. Finally, line 6 builds a graph named AlsGraph based on the input data and graph registration. The overwrite option overwrites a pre-existing graph with the same name.

#### Run ALS algorithm

After graph construction, run the ALS algorithm as follows:



In the example above, the first line calls to the algorithm. The second line specifies which edge property you want to use for the ALS algorithm. Line three specifies which edge label you want to use for this algorithm. Line four specifies the property name for the vertex type, here we use vertex\_type. Line five specifies the property name for edge type, in this case, splits. Line six specifies that at the most we want to run 20 super steps for this algorithm. Line seven configures three feature dimensions for ALS. Line eight sets the convergence threshold to 0. Line nine sets als\_lamda to 0.065. Line ten specifies to output learning at each iteration. Line eleven turns bias calculation on. Line twelve specifies which vertex property names to use for ALS results. Because we configured three feature dimensions: als\_p0, als\_p1, als\_p2, the algorithm will store the results for feature dimension 0, 1, and 2 respectively. Because bias term update is on, als\_bias will store the bias term result.

Depending on your use case, you may want to save your ALS results in one vertex property with a vector value for each vertex, and not in separate vertex properties. We also support this scenario, if you want to do it that way. The example below shows how to use this feature.



The first eleven lines are the same as the previous example. The difference is at Line twelve and Line thirteen. Line twelve enables using a vector as a vertex property value. Line thirteen specifies the property name to use to save the ALS results. In this case, the result will be stored in als\_results in a comma separated list. The bias result will be stored in als\_bias.

The code looks like this:

    Required Parameters:  
    input\_edge\_property\_list : List (comma-separated list of strings)  
        The edge properties which contain the input edge   
        values. If you use more than one edge property, we expect a   
        comma-separated string list.  
    input\_edge\_label : String  
        The edge property which contains the edge label.  
    output\_vertex\_property\_list : List (comma-separated list of strings)  
        The vertex properties which contain the output vertex   
        values. If you use more than one vertex property, we expect a   
        comma-separated string list.  
    vertex\_type : String  
        The vertex property which contains the vertex type.  
    edge\_type : String  
        The edge property which contains edge type.  
  
    num\_mapper : String, optional  
        A reconfigured Hadoop parameter mapred.tasktracker.map.tasks.maximum.  
        Use on the fly when needed for your data sets.  
    mapper\_memory : String, optional  
        A reconfigured Hadoop parameter mapred.map.child.java.opts.   
        Use on the fly when needed for your data sets.  
    vector\_value : String, optional  
        "True" means the algorithm supports a vector as a vertex value.  
        "False" means the algorithm does not support a vector as a vertex value.  
    num\_worker : String, optional  
        The number of Giraph workers.  
        The default value is 15.  
    max\_supersteps : String, optional  
        The number of super steps to run in Giraph.  
        The default value is 10.  
    feature\_dimension : String, optional  
        The feature dimension.  
        The default value is 3.  
    als\_lambda : String, optional  
        The regularization parameter:  
        f = L2\_error + lambda\*Tikhonov\_regularization  
        The default value is 0.065.  
    convergence\_threshold : String, optional  
        The convergence threshold which controls how small the change in   
        validation error must be in order to meet the convergence criteria.  
        The default value is 0.  
    learning\_output\_interval : String, optional  
        The learning curve output interval.  
        The default value is 1.  
        Because each ALS iteration is composed of 2 super steps, the default   
        one iteration means two super steps.  
    max\_val : String, optional  
        The maximum edge weight value.  
        The default value is Float.POSITIVE\_INFINITY.  
    min\_val : String, optional  
        The minimum edge weight value.  
        The default value is Float.NEGATIVE\_INFINITY.  
    bidirectional\_check : String, optional  
        If it is true, Giraph will check whether each edge is bidirectional.  
            The default value is "False".  
    bias\_on : String, optional  
        True means turn bias calculation on, and False means turn bias calculation off.  
        The default value is false.  
    Returns  
    -------  
        output : AlgorithmReport  
        After execution, the algorithm's results are stored in the database. The convergence   
       curve is accessible through the report object.

For a more complete definition of the Lambda parameter, see the [Glossary](#_Lambda).

#### Example

    Graph.ml. als(  
                input\_edge\_property\_list="source",  
                input\_edge\_label=”link",  
                output\_vertex\_property\_list="als\_results, als\_bias",  
                vertex\_type="vertex\_type",  
                edge\_type="edge\_type",  
                num\_worker="3",  
                max\_supersteps="20",  
                feature\_dimension="3"  
                als\_lambda="0.065",  
                convergence\_threshold="0.0",  
                learning\_output\_interval="1",  
                max\_val="5",  
                min\_val="1"  
                bidirectional\_check="false",  
                bias\_on="true"  
    )

### Conjugate Gradient Descent

See: <http://en.wikipedia.org/wiki/Conjugate_gradient_method>.

The Conjugate Gradient Descent (CGD) with Bias for collaborative filtering algorithm.

Our implementation is based on the following paper.  
1) 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](http://public.research.att.com/%7Evolinsky/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.

### Comparison between CGD and ALS

The CGD model is the same as that of ALS except that CGD employs the conjugate gradient descent instead of least squares in optimization. Refer to the ALS discussion above for more details on the model. CGD and ALS share the same bipartite graph representation and the same cost function. The only difference between them is the optimization method.

* ALS solves the optimization problem by least squares that requires a matrix inverse. Therefore, it is computation and memory intensive. 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. CGD converges slower than ALS but requires less memory.
* Whenever 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 Example Usage

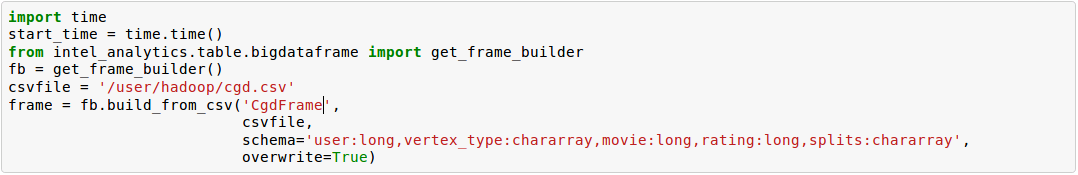
#### Input data format

The CGD algorithm takes input data represented in CSV, JSON or XML format. In this example, we use a CSV file. Each CSV file consists of at least five columns as shown in the table below. The user column is a list of user IDs. The movie column is a list of movie IDs. The rating column records how the user rates the movie in each row. The vertex\_type labels the type of the source vertex in each row. The splits column specifies if this row of data is for training, validation, or testing. We used TR, VA, TE for these three types of splits, respectively.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| user | movie | rating | vertext\_type | splits |
| 1 | -314 | 3 | L | TR |
| 4 | -384 | 4 | L | TR |
| 30 | -371 | 5 | L | TR |
| 435 | -193 | 3 | L | TR |
| 6661 | -304 | 3 | L | TR |
| 6661 | -314 | 4 | L | TR |
| 6661 | -329 | 2 | L | TR |
| 6661 | -332 | 5 | L | TR |
| 6661 |  | 5 | L | TR |

### Data import

To import the CGD data, use the following ipython calls that we provide, as shown below.

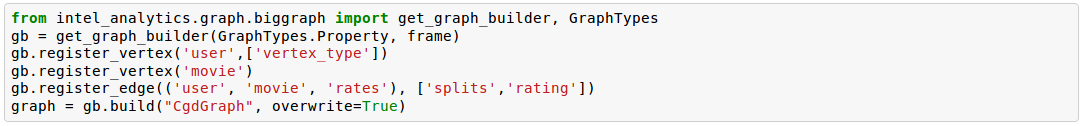


The example above shows how to load CGD input data from a CSV file. The first line imports the related python modules. The second line gets the frame builder into the fb object. The third line specifies the path to the input file. The rest of the lines import the input data. Here is detailed description of the build\_from\_csv method.

* The first argument is the name you want to give to the frame. We use CgdFrame in this example.
* The second argument is the path to your input file, in this case, /user/hadoop/cgd.csv.
* The third argument is the schema of the input data. You need to name each column, and specify the data type of each column in your input CSV input data.
* The fourth argument is whether to overwrite the frame if you have imported data to CGDFrame frame before.

### Graph Construction

After you import the raw data, you register which fields to use for the source vertex, which fields to use for the target vertex, and then construct a graph from your input data.



In the example above, the first line imports the graph construction related python modules. The second line gets the graph builder object into gb. The third to fifth lines register your graph, that is, configure. The third line registers the user column as the source vertex, and registers the vertex\_type vertex property to this vertex. The fourth line registers the movie column as the target vertex. The fifth line registers that each edge from user to movie, with the label rates. Also, rating and splits are two edge properties registered for this algorithm. The sixth line builds a graph based on your input data and graph registration, with graph nameCgdGraph. The overwrite=True in this line means that if you have previously built a graph with the same name, you want to overwrite the old graph.

### Run CGD algorithm

After graph construction, run the CGD algorithm, as shown in the example below.



In the example above, the first line calls the algorithm. The second line specifies which edge property you want to use for the CGD algorithm. The third line specifies which edge label you want to use for this algorithm. Line four specifies the property name for vertex type. We registered vertex\_type for the vertex type above. Line five specifies the property name for edge type. Previously, we registered splits for the edge type. Line six specifies that at most we want to run 20 super steps for this algorithm. Line seven configures three feature dimensions for CGD. Line eight sets the convergence threshold to 0. Line nine sets cgd\_lamda to 0.065. Line ten sets output learning to each iteration. Line eleven turns bias calculation on. Line twelve sets the run to three iterations in each super step. Line thirteen specifies which vertex property names to use for the CGD results. Because we configured three feature dimensions: cgd\_p0, cgd\_p1, and cgd\_p2; CGD will store the results for feature dimension 0, 1, and 2 respectively. Because bias term update is turned on, cgd\_bias will store the bias term result.

Depending on your use case, you may want to save your CGD results in one vertex property with vector values for each vertex, and not in separate vertex properties. We also support this scenario. The example below shows how to use this feature.



The first twelve lines are the same as the previous example. The difference is at lines thirteen and fourteen. Line thirteen enables using vector as a vertex property value. Line fourteen specifies the property name to use to save the CGD results. In this case, the result will be stored in cgd\_results in a comma separated list. The bias result will be stored in cgd\_bias.

    Required parameters:  
    input\_edge\_property\_list : List (comma-separated list of strings)  
        The edge properties which contain the input edge values.   
        If you use more than one edge property. We expect a   
        comma-separated string list.  
    input\_edge\_label : String  
        The edge property which contains the edge label.  
    output\_vertex\_property\_list : List (comma-separated list of strings)  
        The vertex properties which contain the output vertex values.   
        If you use more than one vertex property, we expect a   
        comma-separated string list.  
    vertex\_type : String  
        The vertex property which contains the vertex type.  
    edge\_type : String  
        The edge property which contains the edge type.  
    num\_mapper : String, optional  
        A reconfigured Hadoop parameter mapred.tasktracker.map.tasks.maximum,   
        use on the fly when needed for your data sets.  
    mapper\_memory : String, optional  
        A reconfigured Hadoop parameter mapred.map.child.java.opts,  
        use on the fly when needed for your data sets.  
    vector\_value: String, optional  
        "True" means the algorithm supports a vector as a vertex value.  
        "False" means the algorithm does not support a vector as a vertex value.  
    num\_worker : String, optional  
        The number of Giraph workers.  
        The default value is 15.  
    max\_supersteps :  String, optional  
        The number of super steps to run in Giraph.  
        The default value is 10.  
    feature\_dimension : String, optional  
        The feature dimension.  
        The default value is 3.  
    cgd\_lambda : String, optional  
        The regularization parameter:   
        f = L2\_error + lambda\*Tikhonov\_regularization  
        The default value is 0.065.  
    convergence\_threshold : String, optional  
        The convergence threshold which controls how small the change in validation   
        error must be in order to meet the convergence criteria.  
        The default value is 0.  
    learning\_output\_interval : String, optional  
        The learning curve output interval.  
        The default value is 1.  
        Because each CGD iteration is composed by 2 super steps, the default one   
        iteration means two super steps.  
    max\_val : String, optional  
        The maximum edge weight value.  
        The default value is Float.POSITIVE\_INFINITY.  
    min\_val : String, optional  
        The minimum edge weight value.  
        The default value is Float.NEGATIVE\_INFINITY.  
    bias\_on : String, optional  
        True means turn on bias calculation and False means turn off bias calculation.  
        The default value is false.  
    bidirectional\_check : String, optional  
        If it is true, Giraph will check whether each edge is bidirectional.  
            The default value is "False".  
    num\_iters :   
        The number of CGD iterations in each super step.  
        The default value is 5.  
    After execution, the algorithm's results are stored in database. The convergence   
    curve is accessible through the report object.

#### Example

    Graph.ml.cgd(  
               input\_edge\_property\_list="rating",  
               input\_edge\_label="rates",  
               output\_vertex\_property\_list="cgd\_results, cgd\_bias",  
               vertex\_type="vertex\_type",  
               edge\_type="edge\_type",  
               num\_worker="3",  
               max\_supersteps="20",  
               feature\_dimension="3",  
               cgd\_lambda="0.065",  
               convergence\_threshold="0.001",  
               learning\_output\_interval="1",  
               max\_val="10",  
               min\_val="1",  
               bias\_on="false",  
               num\_iters="3"  
    )

## Graph Analytics

We support three algorithms in this category.

### Average Path Length

The average path length algorithm calculates the average path length from a vertex to any other vertices.

    Parameters  
    ----------  
    input\_edge\_label : String  
        The edge property which contains the edge label.  
    output\_vertex\_property\_list : List (comma-separated list of strings)  
        The vertex properties which contain the output vertex values.  
        If you use more than one vertex property, we expect a comma-separated string list.  
  
    num\_mapper : String, optional  
        A reconfigured Hadoop parameter mapred.tasktracker.map.tasks.maximum.  
        Use on the fly when needed for your data sets.  
    mapper\_memory : String, optional  
        A reconfigured Hadoop parameter mapred.map.child.java.opts.  
        Use on the fly when needed for your data sets.  
    convergence\_output\_interval : String, optional  
        The convergence progress output interval.  
        The default value is 1, which means output every super step.  
    num\_worker : String, optional  
        The number of Giraph workers.  
        The default value is 15.  
  
    Returns  
    -------  
    Output : AlgorithmReport  
        The algorith's results in the database. The progress  
        curve is accessible through the report object.

#### Example

    graph.ml.avg\_path\_len(  
                input\_edge\_label="edge",  
                output\_vertex\_property\_list="apl\_num, apl\_sum",  
                convergence\_output\_interval="1",  
                num\_worker="3"  
    )

### Connected Components

The connected components algorithm finds all connected components in graph. The implementation is inspired by PEGASUS paper.

    Parameters  
    ----------  
    input\_edge\_label : String  
        The edge property which contains the edge label.  
    output\_vertex\_property\_list : List (comma-separated string list)  
        The vertex properties which contain the output vertex values. If you use   
        more than one vertex property, we expect a comma-separated string list.  
  
    num\_mapper : String, optional  
        A reconfigured Hadoop parameter mapred.tasktracker.map.tasks.maximum.  
        Use on the fly when needed for your data sets.  
    mapper\_memory : String, optional  
        A reconfigured Hadoop parameter mapred.map.child.java.opts.  
        Use on the fly when needed for your data sets.  
    convergence\_output\_interval : String, optional  
        The convergence progress output interval.  
        The default value is 1, which means output every super step.  
    num\_worker : String, optional  
        The number of Giraph workers.  
        The default value is 15.  
  
    Returns  
    -------  
    output : AlgorithmReport  
        The algorithm's results in the database. The progress curve is  
        accessible through the report object.

#### Example

    graph.ml.connected\_components(  
                input\_edge\_label="connects",  
                output\_vertex\_property\_list="component\_id",  
                convergence\_output\_interval="1",  
                num\_worker="3"  
    )

### Page Rank

This is the algorithm used by web search engines to rank the relevance of the pages returned by a query.  
See: [http://en.wikipedia.org/wiki/PageRank.](http://en.wikipedia.org/wiki/PageRank)

    Parameters  
    input\_edge\_label : String  
        The edge property which contains the edge label.  
    output\_vertex\_property\_list : List (comma-separated list of strings)  
        The vertex properties which contain the output vertex values.   
        If you use more than one vertex property, we expect a comma-separated string list.  
    num\_mapper : String, optional  
        A reconfigured Hadoop parameter mapred.tasktracker.map.tasks.maximum.  
        Use on the fly when needed for your data sets.  
    mapper\_memory : String, optional  
        A reconfigured Hadoop parameter mapred.map.child.java.opts.  
        Use on the fly when needed for your data sets.  
    num\_worker : String, optional  
        The number of Giraph workers.  
        The default value is 15.  
    max\_supersteps : String, optional  
        The number of super steps to run in Giraph.  
        The default value is 20.  
    convergence\_threshold : String, optional  
        The convergence threshold which controls how small the change in belief value   
        must be in order to meet the convergence criteria.  
        The default value is 0.001.  
    reset\_probability : String, optional  
        The probability that the random walk of a page is reset.  
        The default value is 0.15.  
    convergence\_output\_interval : String, optional  
        The convergence progress output interval.  
        The default value is 1, which means output every super step.  
    Returns  
    -------  
    output : AlgorithmReport  
        The algorithm's results in database. The progress curve is  
        accessible through the report object.

#### Example

    graph.ml.page\_rank(self,  
                      input\_edge\_label="edges",  
                      output\_vertex\_property\_list="page\_rank",  
                      num\_worker="3",  
                      max\_supersteps="20",  
                      convergence\_threshold="0.001",  
                      reset\_probability="0.15",  
                      convergence\_output\_interval="1"  
     )

## Graphical Models

The graphical models find more insights from structured noisy data. We currently support label propagation and loopy belief propagation.

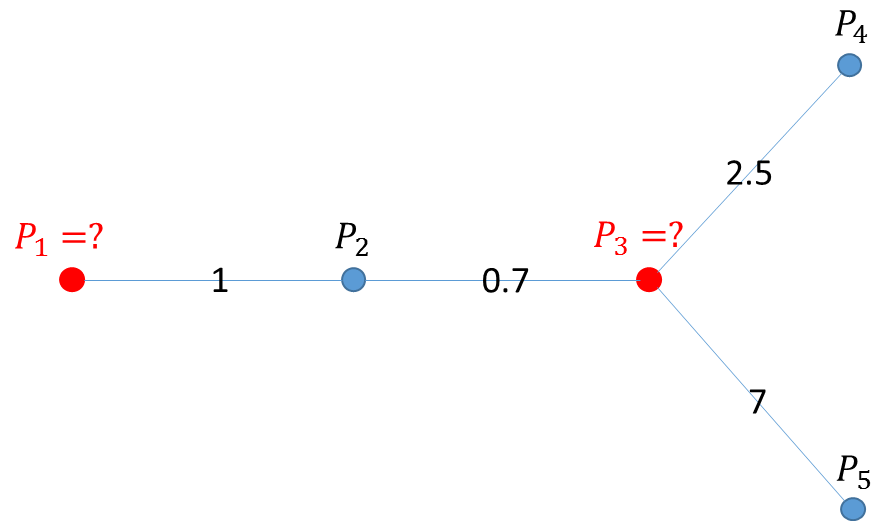
### Label Propagation

Originally proposed as a semi-supervised learning algorithm, label propagation propagates labels from labeled data to unlabeled data along a graph encoding similarity relationships among all data points. It has been used in many classification problems where a similarity measure between instances is available and can be exploited for inference. Specifically, in social network analysis label propagation is used to probabilistically infer data fields that are blank by analyzing data about a user's friends, family, likes and online behavior.

Label Propagation on Gaussian Random Fields. Our implementation is based on this paper:  
1) X. Zhu and Z. Ghahramani. Learning from labeled and unlabeled data with label propagation. Technical Report CMU-CALD-02-107, CMU, 2002.   
See: [http://www.cs.cmu.edu/~zhuxj/pub/CMU-CALD-02-107.pdf](http://www.cs.cmu.edu/%7Ezhuxj/pub/CMU-CALD-02-107.pdf)

### The Lable Propagation Model

A typical representation of the Label Propagation (LP) model is a general graph (see the figure below), where nodes are a set of labeled examples (blue) or unlabeled examples (red) and the edges encode the similarity among examples, such that more similar examples are connected by edges with higher weights. For a labeled example, the label probability Pi is attached to the node. The algorithm then propagates labels from blue nodes on the graph to unlabeled examples (red nodes). The underlying assumption is that similar nodes should have similar labels. The solution can be found with simple matrix operations that iteratively conduct matrix multiplication until convergence.



### LP Example Usage

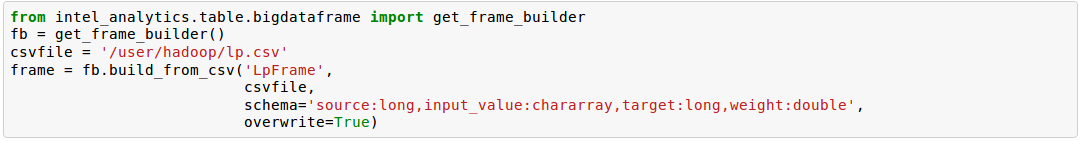
#### Input data format

The Label Propagation (LP) algorithm takes as input data represented in CSV, JSON, or XML format. In this example, we use a CSV file. Each CSV file consists of at least four columns as shown in the example below. The source column is a list of source IDs. The input\_value column is the prior value for the source vertex. In this example, it is vector with two elements in it. The sum of the elements in this vector is 1. The target column is a list of target IDs. The weight column is the weight on the edge from the source to the target.

|  |  |  |  |
| --- | --- | --- | --- |
| **source** | **input\_value** | **target** | **weight** |
| 22 | 0.5 0.5 | 10 | 0.02 |
| 22 | 0.5 0.5 | 19 | 0.51 |
| 110 | 0.5 0.5 | 115 | 0.39 |
| 284 | 0.5 0.5 | 274 | 0.08 |
| 284 | 0.5 0.5 | 275 | 0.11 |
| 284 | 0.5 0.5 | 276 | 0.25 |
| 284 | 0.5 0.5 | 277 | 0.22 |
| 284 | 0.5 0.5 | 278 | 0.32 |
| 284 | 0.5 0.5 | 279 | 0.59 |
| 284 | 0.5 0.5 | 280 | 0.72 |

#### Data import

To import the LP input data, use the following ipython calls we provide.

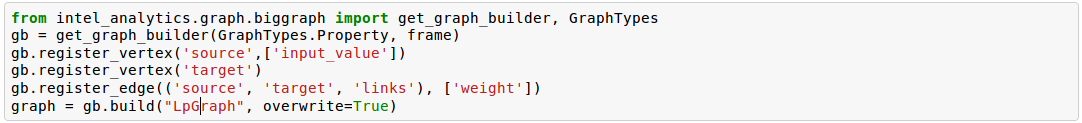


The example above loads the LP input data from the CSV file. The first line imports the related python modules. The second line gets the frame builder into the fb object. The third line specifies the path to the input file. The rest of the lines import the input data. Here is detailed description of the build\_from\_csv method.

* The first argument is the name you want to give to the frame, in this example, LpFrame.
* The second argument is the path to your input file, here /user/hadoop/lp.csv.
* The third argument is the schema for the input data. You need to name each column, and specify the data type of each column in your input CSV input data.
* The fourth argument is whether to overwrite the frame if you have imported data to the LpFrame frame before.

#### Graph Construction

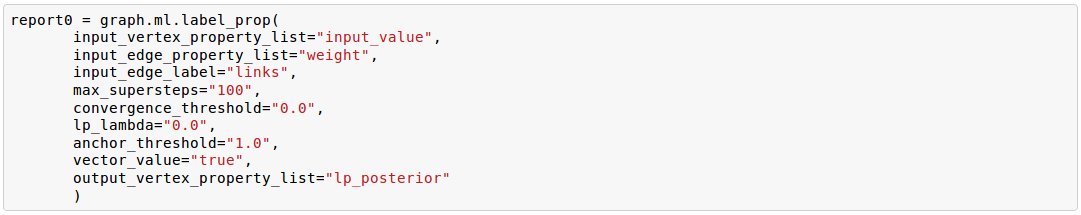
After you import the raw data, you register which fields to use for the source vertex, which fields to use for the target vertex, and then construct a graph from your input data.



In the example above, the first line imports the graph construction related python modules. The second line gets the graph builder object into gb. Lines three to five register your graph. Line three registers the source column as the source vertex, and registers the value vertex property to this vertex. Line four registers the target column as the target vertex. Line five registers each edge from source to target, with the label links. And weight is the edge property registered for this algorithm. Line six builds a graph based on your input data and graph registration, with the graph name LpGraph. The overwrite=True in this line means that if you have previously built a graph with the same name, you want to overwrite the old graph.

#### Run LP algorithm

After graph construction, you are able to run the LP algorithm. Here is an example of it.



In the example above, the first line calls the algorithm. The second line specifies a list of vertex properties where you stored prior values for LP algorithm. In our example, the prior values are stored in the vertex property named input\_value. We also registered this property during graph registration. Line three specifies which edge label you want to use for this algorithm. During graph registration, links was the edge label we registered. Line four specifies that the maximum number of super steps to run is 100. Line five sets the convergence threshold to 0. Line six sets the lamda value for LP to 0. Line seven sets the anchor threshold to 1.0. Line nine specifies which vertex property will be used to store LP results. Because the input prior value for each vertex is a vector with two elements, we expect to see that lp\_posterior is also a vector with two elements.

    Required Parameters  
    input\_vertex\_property\_list : List (comma-separated string list)  
        The vertex properties which contain the prior vertex values.  
        If you use more than one vertex property,   
        we expect a comma-separated string list.  
    input\_edge\_property\_list : List (comma-separated string list)  
        The edge properties which contain the input edge values. If you use   
        more than one edge property, we expect a comma-separated string list.  
    input\_edge\_label : String  
        The edge property which contains the edge label.  
    output\_vertex\_property\_list : List (comma-separated string list)  
        The vertex properties which contain the output vertex values. If you use  
        more than one vertex property, we expect a comma-separated string list.  
    vertex\_type : String  
        The vertex property which contains the vertext type.  
  
    num\_mapper : String, optional  
        A reconfigured Hadoop parameter mapred.tasktracker.map.tasks.maximum.  
        Use on the fly when needed for your data sets.  
    mapper\_memory : String, optional  
        A reconfigured Hadoop parameter mapred.map.child.java.opts.  
        Use on the fly when needed for your data sets.  
    num\_worker : String, optional  
        The number of Giraph workers.  
        The default value is 15.  
    max\_supersteps : String, optional  
        The number of super steps to run in Giraph.  
        The default value is 10.  
    lambda : String, optional  
        The tradeoff parameter: f = (1-lambda)Pf + lambda\*h  
        The default value is 0.  
    convergence\_threshold : String, optional  
        The convergence threshold which controls how small the change in belief  
        value must be in order to meet the convergence criteria.  
        The default value is 0.001.  
    bidirectional\_check : String, optional  
        If set to true, Giraph checks whether each edge is bidirectional.  
        The default value is false.  
    anchor\_threshold : String, optional  
        The anchor threshold [0, 1].  
        Those vertices whose normalized prior values are greater than   
        this threshold will not be updated.  
        The default value is 1.  
  
    Returns  
    -------  
    output : AlgorithmReport  
        The algorithm's results in the database.

#### Example

    graph.ml.label\_prop(  
                input\_vertex\_property\_list="value",  
                input\_edge\_property\_list="weight",  
                input\_edge\_label="links",  
                output\_vertex\_property\_list="lbp\_results",  
                num\_worker="3",  
                max\_supersteps="20",  
                convergence\_threshold="0.0",  
                lp\_lambda="0.0",  
                bidirectional\_check="false",  
                anchor\_threshold="1"  
    )

### Loopy Belief Propagation

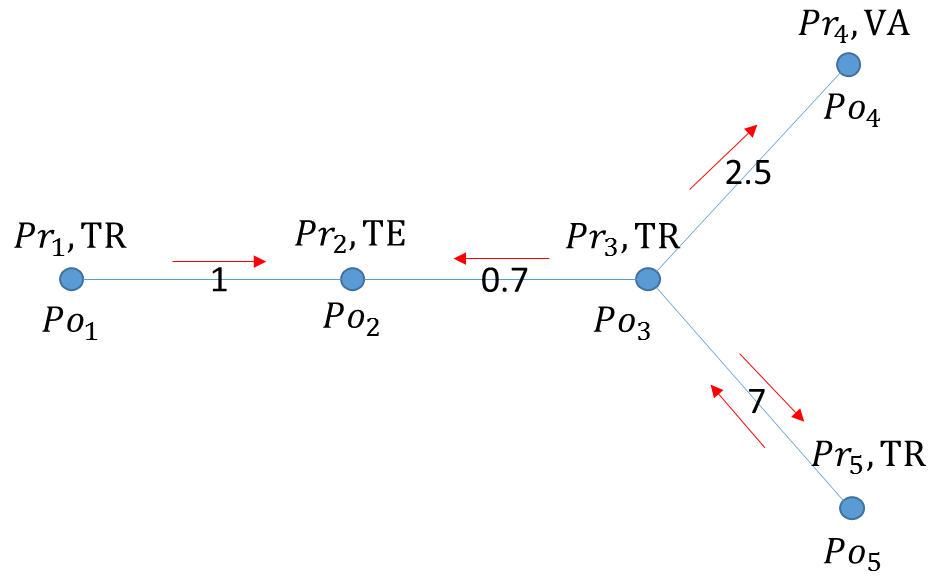
See: <http://en.wikipedia.org/wiki/Belief_propagation>.

This is a message passing algorithm for performing approximate inference on a pair-wise Markov Random Field encoding the classic Potts Model. It 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 similarity relationships between those variables. The algorithm then infers the posterior distribution for each variable by incorporating prior information and graph similarity structure.

The algorithm is described in “K. Murphy, Y. Weiss, and M. Jordan, Loopy-belief Propagation for Approximate Inference: An Empirical Study, UAI 1999”. (<http://arxiv.org/ftp/arxiv/papers/1301/1301.6725.pdf>) We also extended it to support training, validation and test, a common practice in machine learning.

### The Loopy Belief Propagation Model

The algorithm performs approximate inference on an undirected graph of hidden variables, where each variable is represented as a node, and edges encode the similarity of it to its neighbors. Initially a prior noisy prediction Pri is attached to each node, then the algorithm infers the posterior distribution Poi of each node by propagating and collecting messages to and from its neighbors and updating the beliefs. Even though its convergence is not guaranteed for graphs containing loops, LBP has demonstrated empirical success in many areas, especially in turbo codes, and in practice these approximations often perform well.



The original Loopy Belief Propagation (LBP) algorithm was designed for approximate inference in general graphs. It lacks functionality to make predictions for nodes in hold-out sets that are often excluded from the training graph. Inserting hold-out sets into training graph and launching LBP would have an undesirable impact, because in this case the hold-out sets would participate in the training or inference. To support the training, validation, and test scenario, we extend the original LBP algorithm by annotating each node as “TR”, “VA”, or “TE” (see the figure above). For “TR” nodes, their behavior is the same as before, for example, propagating or collecting messages to or from their neighbors at each iteration. But for “VA” and “TE” nodes, they cannot propagate any message to their neighbors, but can only collect messages (see the red arrows representing message passing in the figure above). This makes “VA” and “TE” nodes great listeners but not messengers, and therefore mitigates their impact on the training process. Annotating all nodes as “TR” causes the algorithm to degenerate to the original LBP.

Loopy belief propagation on Markov Random Fields(MRF). This algorithm was originally designed for acyclic graphical models, then it was found that the Belief Propagation algorithm can be used in general graphs. The algorithm is then sometimes called "loopy" belief propagation, because graphs typically contain cycles, or loops. In Giraph, we run the algorithm in iterations until it converges.

    Required Parameters  
    input\_vertex\_property\_list : List (comma-separated list of strings)  
        The vertex properties which contain prior vertex values. If you use   
        more than one vertex property, we expect a comma-separated string list.  
    input\_edge\_property\_list : List (comma-separated list of strings)  
        The edge properties which contain the input edge values. If you use more   
        than one edge property, we expect a comma-separated string list.  
    input\_edge\_label : String  
        The edge property which contains the edge label.  
    output\_vertex\_property\_list : List (comma-separated list of strings)  
        The vertex properties which contain the output vertex values. If you use more   
        than one vertex property, we expect a comma-separated string list.  
    num\_mapper : String, optional  
        A reconfigured Hadoop parameter mapred.tasktracker.map.tasks.maximum.  
        Use on the fly when needed for your data sets.  
    mapper\_memory : String, optional  
        A reconfigured Hadoop parameter mapred.map.child.java.opts.  
        Use on the fly when needed for your data sets.  
    num\_worker : String, optional  
        The number of Giraph workers.  
        The default value is 15.  
    max\_supersteps : String, optional  
        The number of super steps to run in Giraph.  
        The default value is 10.  
    smoothing : String, optional  
        The Ising smoothing parameter.  
        The default value is 2.  
    convergence\_threshold : String, optional  
        The convergence threshold which controls how small the change in validation error must be   
        in order to meet the convergence criteria.  
        The default value is 0.001.  
    bidirectional\_check : String, optional  
        If set to true, Giraph checks whether each edge is bidirectional.  
        The default value is false.  
    anchor\_threshold : String, optional  
        The anchor threshold [0, 1].  
        Those vertices whose normalized prior values are greater than this threshold   
        will not be updated.  
        The default value is 1.  
  
    Returns  
    -------  
    output : AlgorithmReport  
        The algorithm's results in the database.

#### Example

    graph.ml.belief\_prop(  
                        input\_vertex\_property\_list="values",  
                        input\_edge\_property\_list="weight",  
                        input\_edge\_label="links",  
                        output\_vertex\_property\_list="lbp\_results",  
                        num\_worker="3",  
                        max\_supersteps="10",  
                        convergence\_threshold="0.0001",  
                        smoothing="2",  
                        bidirectional\_check="false",  
                        anchor\_threshold="1"

    )

For a more complete definition of the Ising Smoothing parameter, see the [Glossary](#_Ising_Smoothing_Parameter).

## Topic Modeling

For Topic Modeling, see: <http://en.wikipedia.org/wiki/Topic_model>

### Latent Dirichlet Allocation

We currently support Latent Dirichlet Allocation (LDA) for our topic modeling.

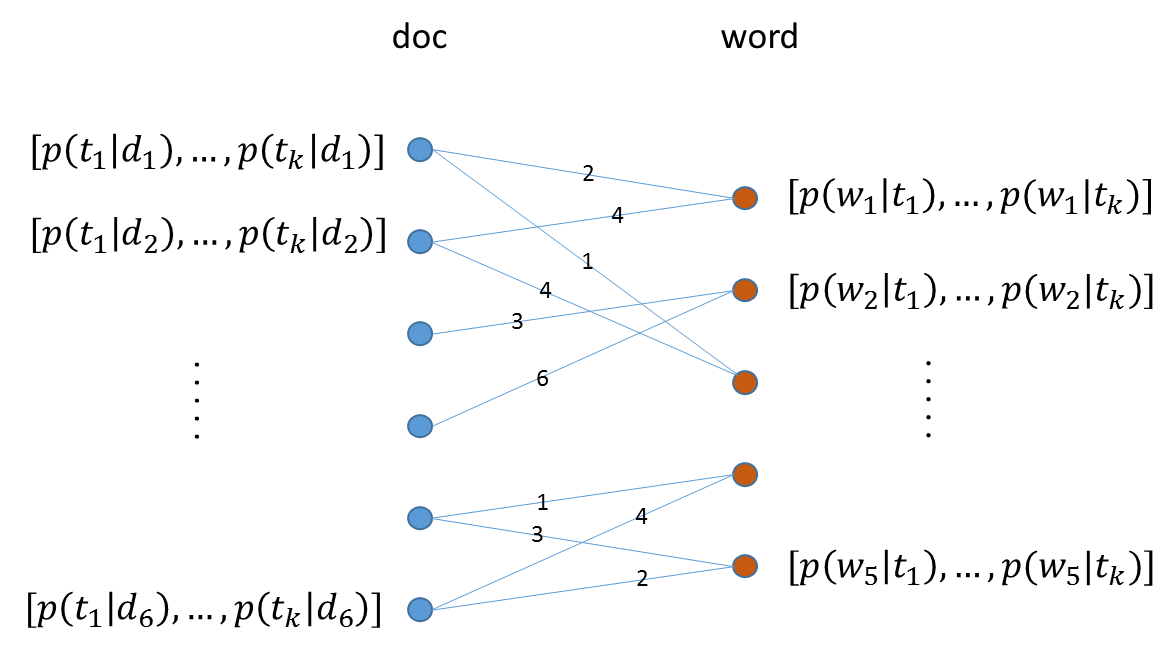
See: <http://en.wikipedia.org/wiki/Latent_Dirichlet_allocation>

This is an algorithm for topic modeling that discovers the hidden topics from a collection of documents and annotates the document according to those topics. You can use resulting topical representation as a feature space in information retrieval tasks to group topically related words and documents and to organize, summarize and search the texts. See the excellent demo of LDA on Wikipedia here:  
[http://www.princeton.edu/~achaney/tmve/wiki100k/browse/topic-presence.html](http://www.princeton.edu/%7Eachaney/tmve/wiki100k/browse/topic-presence.html)

Solving the latent topic assignment problem is an NP-Hard task. There exist several approximate inference algorithms. Our implementation is based on the CVB0 LDA algorithm, one of the state of the art LDA solvers, presented in “Y.W. Teh, D. Newman, and M. Welling, A Collapsed Variational Bayesian Inference Algorithm for Latent Dirichlet Allocation, NIPS 19, 2007”.  
http://www.gatsby.ucl.ac.uk/~ywteh/research/inference/nips2006.pdf

#### The LDA Model

A typical representation of LDA is a bipartite graph, where nodes on the left side represent a collection of documents and nodes on the right side represents a set of words (for example., vocabulary), and edges encode number of occurrences of a word in a corresponding document (see the example below).



**The LDA Algorithm**

After the execution of LDA on the input bi-partite graph, each node in the graph will be associated with a vector of length k (such as, the number of topics specified by user). For a document node d, *p(ti|d)* denotes the distribution over topics to document *d*, and   
. For a word node *w*, *p(w|ti)* denotes the distribution over words to each topic *ti*. Theoretically, *p(w|ti)* should be normalized such that . But this normalization is ignored in the implementation because it requires normalizing scores across all the words, which incurs an additional map-reduce step. This normalization is expensive but wouldn’t bring us too much benefit because to identify the top words for a topic we only need a sort across all the words.

At a high-level, LDA extracts semantically similar words into a topic, such as “foods,” “sports,” and “geography,” and it groups similar documents according to the extracted topics. The underlying assumptions are intuitive: 1, words in the same documents are topically related; (2) documents that share common words are likely about similar topics.

### LDA Example Usage

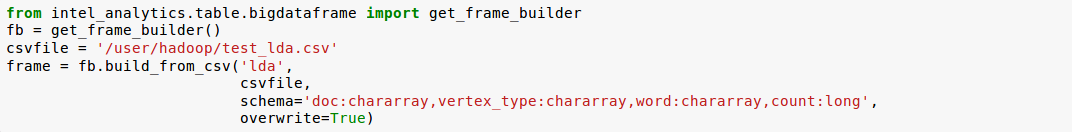
#### Input data format

The LDA algorithm takes an input text corpus represented in CSV, JSON or XML format. We use a CSV file in this example. Each CSV file consists of at least four columns as shown in the table below. The “doc” column is a list of document titles. The “word” column is a list of words in these documents. The “count” column records how many times a word appears in a given document. The “vertex\_type” labels the type of the source vertex in each row.

|  |  |  |  |
| --- | --- | --- | --- |
| **doc** | **word** | **count** | **vertex\_type** |
| A E Housman | reflist | 1 | L |
| A E Housman | bgcolor | 1 | L |
| A E Housman | head | 3 | L |
| A E Housman | relates | 1 | L |
| AK47 | peace | 1 | L |
| Akira Kurosawa | unravel | 1 | L |
| Condensed matter physics | However | 4 | L |
| Dolphin | atlantis | 1 | L |
| Dolphin | drowning | 1 | L |
| Dolphin | just | 2 | L |

#### Data import

To import the LDA input data, you can use the following iPython calls:

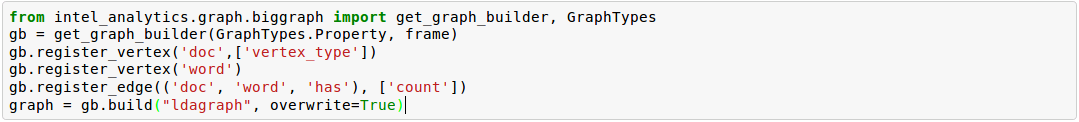


The example above loads the LDA input data from a CSV file. The first line imports the python modules. The second line gets the frame builder into the fb object. The third line specifies where the path to the input file. The remainder of the lines perform the data import through the build\_from\_csv method:

* The first argument is a name you want to give to the frame. This example uses lda.
* The second argument the path to your input file. In this case: /user/hadoop/test\_lda.csv.
* The third argument is the schema of the input data. You need to name each column, and specify the data type of each column in your input CSV input data.
* The fourth argument is whether to overwrite the frame; true overwrites the frame, if you have imported data to the lda frame before.

#### Graph Construction

After you import the raw data, you register which fields to use for the source vertex, which fields to use for the target vertex, and then construct a graph from your input data.



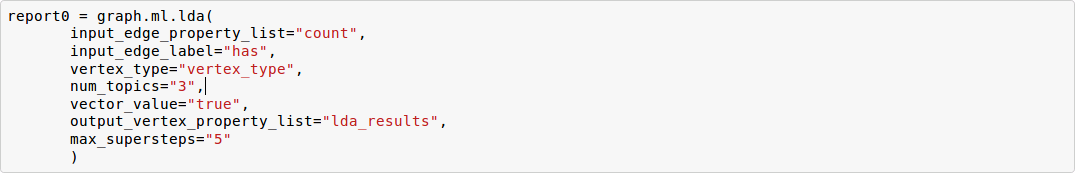
In the example above, the first line imports the python modules needed for graph construction. The second line gets the graph builder object into gb. The third to fifth lines register the graph. Line 3 registers the doc column as the source vertex, and registers the vertex property vertex\_type to this vertex. Line 4 registers the word column as the target vertex, and line 5 registers an edge from doc to word, with the label has, and count as the edge property. Finally, line 6 builds a graph named ldagraph based on the input data and graph registration. The overwrite option specifies that an existing graph with this name will be overwritten.

#### Run LDA algorithm

After graph construction, we run the LDA algorithm as shown:

In example above, the first line starts the call to the algorithm. The second and third lines specify the edge property and edge label to use. Line 4 specifies the property name for the vertex type; in this example we register the property named vertex\_type. The fifth line sets the num\_topics parameter used by LDA. Line six specifies the vertex property names in which to save the LDA results; because we configure three topics, these three properties will store the normalized probability that the vertex belongs to topics 0, 1, and 2 respectively. Finally, line seven specifies that we want to run at most five super steps for this algorithm.

It is possible to save the LDA results either in separate vertex properties, or in one vertex property with vector value for each vertex. The example below shows this feature.



The first five lines are the same as the previous example. The difference is at the sixth and seventh lines. Line six enables using a vector as a vertex property value while line seven specifies the property name to use to save the LDA results. In this case, the result will be stored in a comma separated list. The eighth line is the same as the seventh line in previous example.

new

    Parameters  
    ----------  
    input\_edge\_property\_list : List (comma-separated list of strings)  
        The edge properties which contain the input edge values. If you use   
        more than one edge property, we expect a comma-separated string list.  
    input\_edge\_label : String  
        The edge property which contains the edge label.  
    output\_vertex\_property\_list : List (comma-separated list of strings)  
        The vertex properties which contain the output vertex values. If you use more than   
        one vertex property, we expect a comma-separated string list.  
    vertex\_type : String  
        The vertex property which contains the vertex type.  
  
    num\_mapper : String, optional  
        A reconfigured Hadoop parameter mapred.tasktracker.map.tasks.maximum.  
        Use on the fly when needed for your data sets.  
    mapper\_memory : String, optional  
        A reconfigured Hadoop parameter mapred.map.child.java.opts.   
        Use on the fly when needed for your data sets.  
    vector\_value : String, optional  
        "True" means the algorithm supports a vector as a vertex value.  
        "False" means the algorithm does not support a vector as a vertex value.  
    num\_worker : String, optional  
        The number of workers.  
        The default value is 15.  
   max\_supersteps :String, optional  
        The number of super steps to run in Giraph.  
        The default value is 20.  
    alpha : String, optional  
        The document-topic smoothing parameter.  
        The default value is 0.1.  
    beta : String, optional  
        The term-topic smoothing parameter.  
        The default value is 0.1.  
    convergence\_threshold : String, optional  
        The convergence threshold which controls how small   
        the change in edge value must be in order to meet the   
        convergence criteria.  
        The default value is false.  
    evaluate\_cost : String, optional  
        True means turn cost evaluation on, and False means   
        turn cost evaluation off.  
        The default value is false.  
    max\_val : String, optional  
        The maximum edge weight value.  
        The default value is Float.POSITIVE\_INFINITY.  
    min\_val : String, optional  
        The minimum edge weight value.  
        The default value is Float.NEGATIVE\_INFINITY.  
    num\_topics : String, optional  
        The number of topics to identify.  
        The default value is 10.   
    Returns  
    -------  
    output : AlgorithmReport  
        The algorithm's results in the database. The convergence curve is  
        accessible through the report object.

#### Example

    graph.ml.lda(  
                input\_edge\_property\_list="frequency",  
                input\_edge\_label="has",  
                output\_vertex\_property\_list="lda\_results",  
                vertex\_type="vertex\_type",  
                edge\_type="edge\_types",  
                num\_worker="3",  
                max\_supersteps="20",  
                alpha="0.1",  
                beta="0.1",  
                convergence\_threshold="0.0001",  
                evaluate\_cost="true",  
                max\_val=" Float.POSITIVE\_INFINITY",  
                min\_val=" Float.NEGATIVE\_INFINITY",  
                num\_topics="10"  
    )

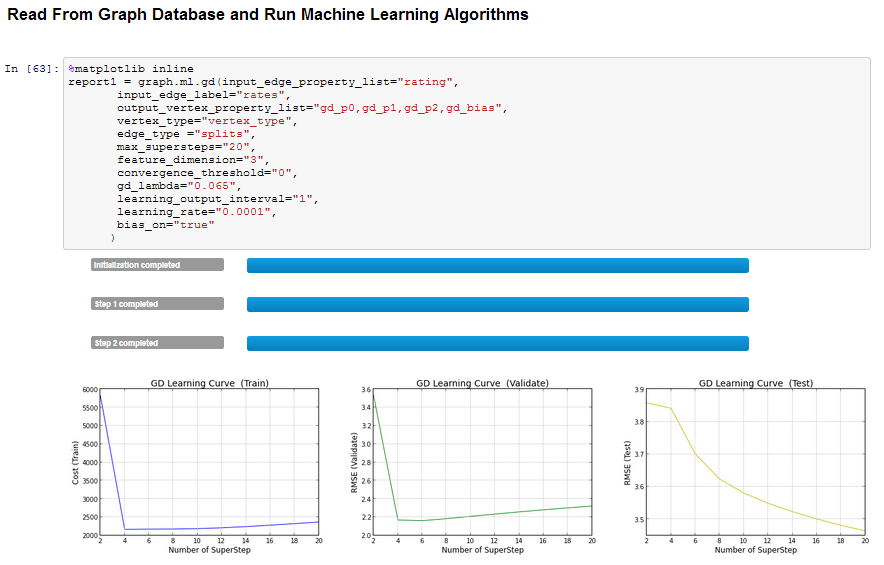
## Perform Analytics on the Graph

When you have all your data in the Titan graph database, you are now able to perform additional analytics to view and explore your data. This is where you look at what was once raw data and now has some form and much more information. You will use the Machine Learning API calls to do this. See the [Machine Learning](#_Machine_Learning) page for more details.

We built the interface in iPython notebooks, because many data scientists are already familiar with the Python interface and use of iPython notebooks. For complex graph traversals and mutation operations, see: <https://github.com/thinkaurelius/titan/wiki/Gremlin-Query-Language>.

You do not have to read the graph. Once you have the graph object, from the graph construction section, you can run Machine Learning algorithms on it immediately. The first line of the code in Figure 1, prepares iPython for the upcoming visualizations.

In the second line, we name our report. In this case, report1, but you can name it whatever you want. The graph.ml.gd() method is the gradient descent algorithm. You can call the other algorithms in the same way, such as: graph.ml.als() for the alternating least squares algorithm (using the appropriate parameters, as described in the API documentation). In the graph.ml.gd() method call, we assign each of the parameters a value. Refer to the API documentation and the [Machine Learning Algorithms](#_Machine_Learning_Algorithms) page.



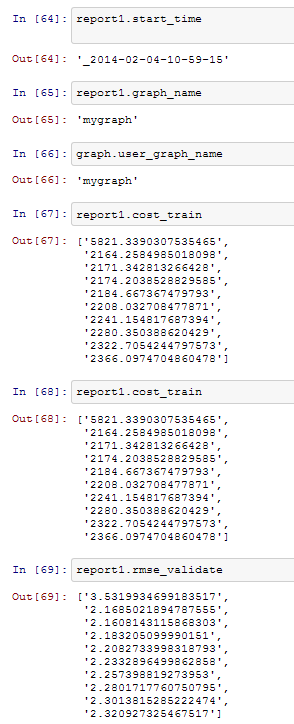
**Figure 1: Read from Graph Database and Run Machine Learning Algorithms.**

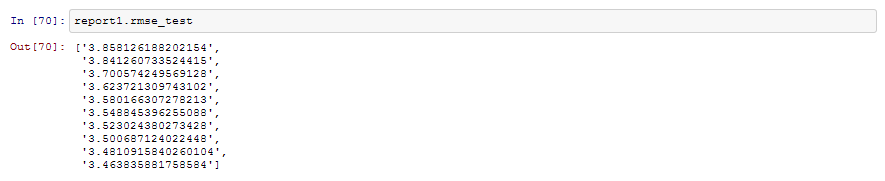
After the algorithm has finished, you can use the report object to look at how the execution has performed.

In Figure 2, below, in line 64, we can view the start time so we can keep track of how long this takes.

In line 65, we can see the assigned graph name in report1, which is the underlying name of the Titan graph, that the algorithm has been run.

In line 67, you can see how the algorithm has performed and how with each iteration the cost has improved.

****

****

**Figure 2, Graph Creation**

In line 69 above, rmse\_validate is a command that shows the root-mean-square error in each of the iterations on the validation data set.

Now you need to run the algorithm against the test data set to see how it performs using the data set aside for testing purposes.

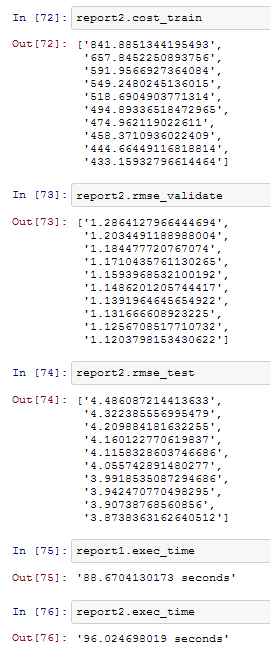
In line 70, rmse\_test determines the root-mean-square error on the test data.

In line 71 below, the graph.ml.als() command runs the alternating least squares algorithm on the same dataset.



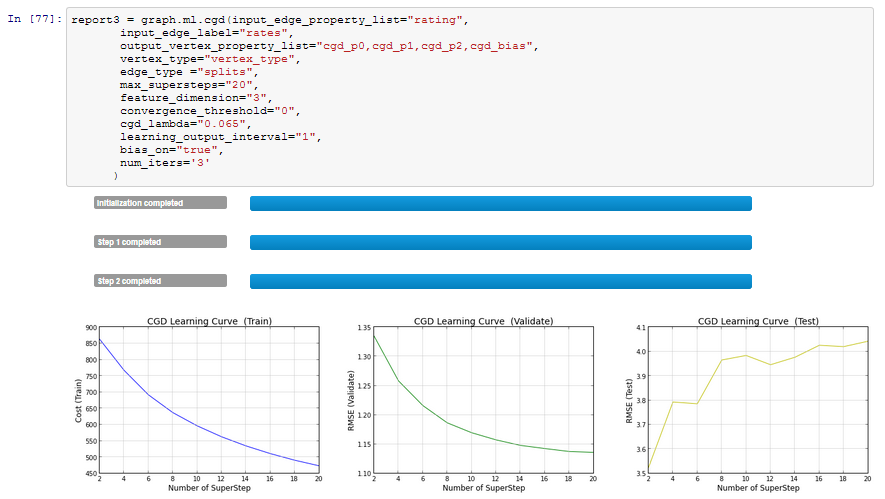
**Figure 3, Run Alternating Least Squares Algorithm**

Once again, you can see the results of the ALS algorithm and how it performed.



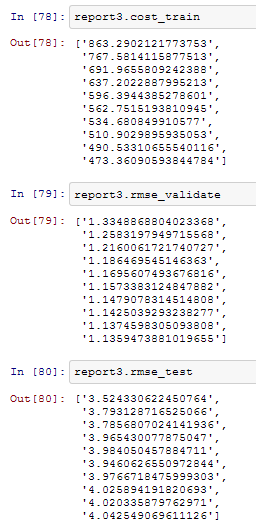
**Figure 4, Cost Training, Validation, and Testing**

Now we run the conjugated gradient descent algorithm on the same data set.



**Figure 5, Run the Conjugated Gradient Descent Algorithm**

The last commands you can run for this part of analytics are looking at the runs.



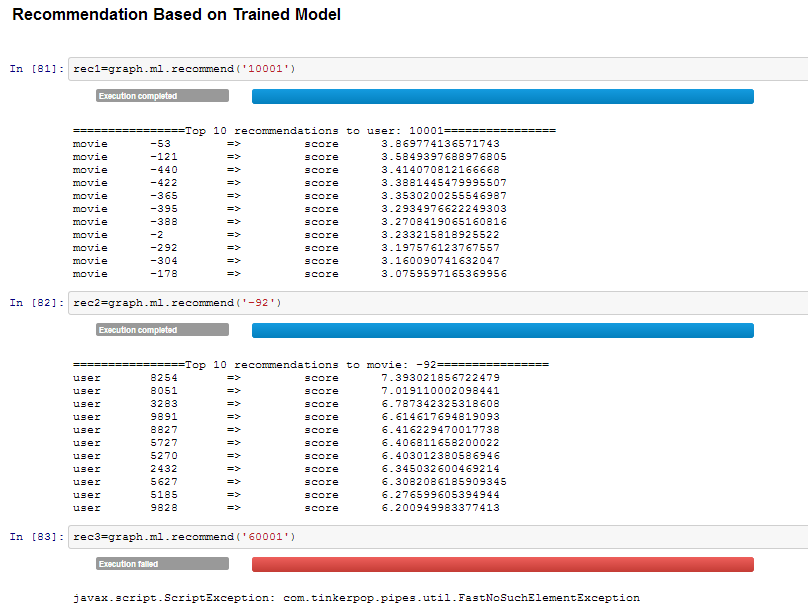
**Figure 6, Cost, Validate, and Test**

As you can see from the examples above, the Intel Data Platform: Analytics Toolkit makes data transformations and running prebuilt algorithms easier and faster with the simple Python interface.

This last figure shows a recommendation based on trained learning. We look at the recommendation for a user, in this case, 10001, and what the top 10 recommended movies and ratings are for that user.

For movie '-292,' the recommendation shows what are the top 10 users and their scores that will most enjoy this movie.

Finally, we deliberately entered an unknown value to the recommendation as an example of what our errors look like.



**Figure 7, Trained Learning and Error Message**

# Glossary

#### Alternating Least Squares

The "Alternating Least Squares with Bias for collaborative filtering algorithms" is an algorithm used by the Intel Data Platform: Analytics Toolkit. You can find out more about it here: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.173.2797> and here: [http://public.research.att.com/~volinsky/netflix/kdd08koren.pdf](http://public.research.att.com/%7Evolinsky/netflix/kdd08koren.pdf) (see equation 5).

#### Average Path Length

“Average path length is a concept in network topology that is defined as the average number of steps along the shortest paths for all possible pairs of network nodes. It is a measure of the efficiency of information or mass transport on a network.” Lifted from: <http://en.wikipedia.org/wiki/Average_path_length>.

#### Belief Propagation

See [Loopy Belief Propagation](#_Loopy_Belief_Propagation).

#### Beysian Network

Borrowed from Wikipedia. "A Beysian Network is a probabilistic graphic model that represents a set of random variables and their conditional dependencies through a directed acyclic graph (DAG). For example, a Bayesian network could represent the probabilistic relationships between diseases and symptoms. Given symptoms, the network can be used to compute the probabilities of the presence of various diseases." Contrast with [Markov Random Fields](#_Markov_Random_Fields). See <http://en.wikipedia.org/wiki/Bayesian_network>.

#### Bias-variance tradeoff

Lifted from: <http://en.wikipedia.org/wiki/Bias_variance#Bias-variance_tradeoff>. Main article: Bias-variance dilemma (<http://en.wikipedia.org/wiki/Bias-variance_dilemma>)

A first issue is the tradeoff between bias and variance. Imagine that we have available several different, but equally good, training data sets. A learning algorithm is biased for a particular input x if, when trained on each of these data sets, it is systematically incorrect when predicting the correct output for x. A learning algorithm has high variance for a particular input x if it predicts different output values when trained on different training sets. The prediction error of a learned classifier is related to the sum of the bias and the variance of the learning algorithm. Generally, there is a tradeoff between bias and variance. A learning algorithm with low bias must be "flexible" so that it can fit the data well. But if the learning algorithm is too flexible, it will fit each training data set differently, and hence have high variance. A key aspect of many supervised learning methods is that they are able to adjust this tradeoff between bias and variance (either automatically or by providing a bias/variance parameter that the user can adjust).

#### Bias vs Variance

In this context, “bias” means accuracy, while “variance” means accounting for outlier data points.

#### Congugate Gradient Descent

The "Congugate Gradient Descent with Bias for Collaborative Filtering algorithm is an algorithm used by the Intel Data Platform: Analytics Toolkit. You can find out more about it here: [http://public.research.att.com/~volinsky/netflix/kdd08koren.pdf](http://public.research.att.com/%7Evolinsky/netflix/kdd08koren.pdf) (see equation 5).

#### Convergence

Where a calculation (often an iterative calculation) reaches a certain value. See [http://en.wikipedia.org/wiki/Convergence\_(mathematics)](http://en.wikipedia.org/wiki/Convergence_%28mathematics%29).

#### Directed Acyclic Graph (DAG)

Lifted from Wikipedia. In mathematics and computer science, a directed acyclic graph (DAG), is a directed graph with no directed cycles. That is, it is 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 Graphs](#_Undirected_Graph). See <http://en.wikipedia.org/wiki/Directed_acyclic_graph>.

#### Edge

An edge is the link between two vertices in a graph database. Edges can have direction, or be undirected. Edges are said to have a source and a destination, usually meaning the vertex to the left and the vertex to the right. Each edge has a label, which is the edge's unique name, and a property map. The property map may contain 0 or more properties. An edge can be uniquely identified from its source, destination, and label. See [vertex](#_Vertex_(Vertices)) in this glossary, and: <https://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model>.

#### ETL – Extract, Transform, and Load

Lifted from Wikipedia. In computing, extract, transform, and load (ETL) 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. See <http://en.wikipedia.org/wiki/Extract,_transform,_load>.

#### Gaussian Random Fields

Borrowed from Wikipedia. A Gaussian random field (GRF) is a random field involving Gaussian probability density functions of the variables. A one-dimensional GRF is also called a Gaussian process.

One way of constructing a GRF is by assuming that the field is the sum of a large number of plane, cylindrical, or spherical waves with uniformly distributed random phase. Where applicable, the central limit theorem dictates that at any point, the sum of these individual plane-wave contributions will exhibit a Gaussian distribution. This type of GRF is completely described by its power spectral density, and hence, through the Wiener-Khinchin theorem, by its two-point autocorrelation function, which is related to the power spectral density through a Fourier transformation. For details on the generation of Gaussian random fields using Matlab, see the circulant embedding method for Gaussian random field. See <http://en.wikipedia.org/wiki/Gaussian_random_field>.

#### Graph

In mathematics, and more specifically in graph theory, a graph is a representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by mathematical abstractions called vertices, and 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. See: [http://en.wikipedia.org/wiki/Graph\_(mathematics)](http://en.wikipedia.org/wiki/Graph_%28mathematics%29).

#### 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.
* outer: A union of indexes.
* 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

Gremlin is a graph query language, akin to SQL, that enables users to manipulate and query a graph. Gremlin works with the [Titan Graph Database](http://thinkaurelius.github.io/titan/), though it is made by a different company. See <https://github.com/tinkerpop/gremlin/wiki>.

#### Ising Smoothing Parameter

The smoothing parameter in the Ising model. See: <http://en.wikipedia.org/wiki/Ising_model>. 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.

#### Labeled Data vs Unlabeled Data

Borrowed from Wikipedia. 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. See: <http://en.wikipedia.org/wiki/Machine_learning#Algorithm_types>.

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. See <http://en.wikipedia.org/wiki/Semi-supervised_learning>.

#### Lambda

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. (Adapted from: <http://openclassroom.stanford.edu/MainFolder/DocumentPage.php?course=MachineLearning&doc=exercises/ex5/ex5.html>.)

#### Latent Dirichlet Allocation

Borrowed from Wikipedia. In natural language processing, latent Dirichlet allocation (LDA) is 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. See <http://en.wikipedia.org/wiki/Latent_Dirichlet_allocation>.

#### Loopy Belief Propagation

Belief Propagation is an algorithm that makes inferences on graph models, like a Bayesian network or Markov Random Fields. It is called Loopy when the algorithm runs iteratively until convergence. See <http://en.wikipedia.org/wiki/Loopy_belief_propagation>.

#### 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.

#### MapReduce

MapReduce is a programming model for processing large data sets with a parallel, distributed algorithm on a cluster.

A MapReduce program 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. See <http://en.wikipedia.org/wiki/Map_reduce>.

#### Markov Random Fields (MRF)

Markov Random fields, or Markov Network, are an undirected graph model that may be cyclic. This contrasts with [Beysian Networks](#_Beysian_Network), which are directed and acyclic. See <http://en.wikipedia.org/wiki/Markov_random_field>.

#### Page Rank

The PageRank algorithm, used to rank web pages in a web search. See: <http://en.wikipedia.org/wiki/PageRank>.

#### Property Map

A property map is a key-value map. Both edges and vertices have property maps. See: <https://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model>.

#### RDF (Resource Description Framework)

The Resource Description Framework (RDF) is a family of World Wide Web Consortium (W3C) specifications originally designed as a metadata data model. It has come to be used as a general method for conceptual description or modeling of information that is implemented in web resources, using a variety of syntax notations and data serialization formats. See: <http://en.wikipedia.org/wiki/Resource_Description_Framework>.

#### 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. See Supervised Learning, and Unsupervised Learning.

#### 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.” See <http://en.wikipedia.org/wiki/Smoothing> and [http://en.wikipedia.org/wiki/Relaxation\_(iterative\_method)](http://en.wikipedia.org/wiki/Relaxation_%28iterative_method%29).

#### 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. See Unsupervised Learning, and Semi-Supervised Learning.

#### 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 n(n - 1)/2. Contrast with [Directed Acyclic Graph](#_Directed_Acyclic_Graph). See also: <http://en.wikipedia.org/wiki/Undirected_graph#Undirected_graph>.

#### 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. See Supervised Learning, and Semi-Supervised Learning.

#### Vertex (Vertices)

A vertex is a data point in a graph database. 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. See [Edge](#_Edge) in this glossary, and: <https://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model>.