

Table of Contents

Introduction	0
Getting started	1
Working with data	2
Tabular data	2.1
Loading and Saving	2.1.1
Data Manipulation	2.1.2
Spark RDDs	2.1.3
SQL Databases	2.1.4
Graph data	2.2
Time series data	2.3
Visualization	2.4
Feature Engineering	2.5
Numeric Features	2.5.1
Quadratic Features	2.5.1.1
Feature Binning	2.5.1.2
Numeric Imputer	2.5.1.3
Categorical Features	2.5.2
One Hot Encoder	2.5.2.1
Count Thresholder	2.5.2.2
Categorical Imputer	2.5.2.3
Count Featurizer	2.5.2.4
Text Features	2.5.3
TF-IDF	2.5.3.1
Tokenizer	2.5.3.2
RareWordTrimmer	2.5.3.3
BM25	2.5.3.4
PartOfSpeechExtractor	2.5.3.5
SentenceSplitter	2.5.3.6
Image Features	2.5.4
Deep Feature Extractor	2.5.4.1

Other Transformations	2.5.5
Hasher	2.5.5.1
Random Projection	2.5.5.2
Transformer Chain	2.5.5.3
Custom Transformer	2.5.5.4
Modeling data	3
Classification	3.1
Logistic Regression	3.1.1
Nearest Neighbor Classifier	3.1.2
SVM	3.1.3
Decision Tree Classifier	3.1.4
Random Forest Classifier	3.1.5
Boosted Trees Classifier	3.1.6
Neuralnet Classifier	3.1.7
Regression	3.2
Linear Regression	3.2.1
Decision Tree Regression	3.2.2
Boosted Trees Regression	3.2.3
Random Forest Regression	3.2.4
Advanced Deep Learning with MXNet	3.3
Graph analytics	3.4
Examples	3.4.1
Clustering	3.5
KMeans	3.5.1
DBSCAN	3.5.2
Nearest Neighbors	3.6
Text analysis	3.7
Processing text	3.7.1
Topic models	3.7.2
Evaluating Models	3.8
Regression Metrics	3.8.1
Classification Metrics	3.8.2
Model parameter search	3.9
Models	3.9.1

Choosing a search space	3.9.2
Evaluation functions	3.9.3
Distributed execution	3.9.4
Applications	4
Recommender systems	4.1
Using trained models	4.1.1
Choosing a model	4.1.2
Data matching	4.2
Record Linker	4.2.1
Deduplication	4.2.2
Autotagger	4.2.3
Similarity Search	4.2.4
Lead Scoring	4.3
Churn prediction	4.4
Using a trained model	4.4.1
Alternate input formats	4.4.2
How it works	4.4.3
Frequent Pattern Mining	4.5
Sentiment analysis	4.6
Applying a sentiment classifier	4.6.1
Product sentiment analysis and review data	4.6.2
Anomaly Detection	4.7
Local Outlier Factor	4.7.1
Moving Z-Score	4.7.2
Bayesian Changepoints	4.7.3
Turi Distributed	5
Asynchronous Jobs	5.1
Installing on Hadoop	5.2
Clusters	5.3
End-to-End Example	5.4
Distributed Job Execution	5.5
Distributed Machine Learning	5.6
Monitoring Jobs	5.7

Session Management	5.8
Dependencies	5.9
Turi Predictive Services	6
Conclusion	7
Exercises	8
Tabular data	8.1
Graph data	8.2
Graph analytics	8.3
Classification	8.4
Text analysis	8.5
Recommender systems	8.6
FAQ/Common Problems	9
Contributing	10

Turi Machine Learning Platform User Guide

Our mission at Turi is to build the most powerful and usable data science tools that enable you to go quickly from inspiration to production.

[GraphLab Create](#) is a Python package that allows programmers to perform end-to-end large-scale data analysis and data product development.

- **Data ingestion and cleaning with SFrame**. SFrame is an efficient disk-based tabular data structure that is not limited by RAM. This lets you scale your analysis and data processing to handle terabytes of data, even on your laptop.
- **Data exploration and visualization with GraphLab Canvas**. GraphLab Canvas is a browser-based interactive GUI that allows you to explore tabular data, summary plots and statistics.
- **Network analysis with SGraph**. SGraph is a disk-based graph data structure that stores vertices and edges in SFrames.
- **Predictive model development with machine learning toolkits**. GraphLab Create includes several toolkits for quick prototyping with fast, scalable algorithms.
- **Production automation with data pipelines**. Data pipelines allow you to assemble reusable code tasks into jobs and automatically run them on common execution environments (e.g. Amazon Web Services, Hadoop).

In this guide, you will learn how to use GraphLab Create to:

- munge and explore both structured and unstructured data
- use advanced machine learning methods to build predictive models and recommender systems
- put your code into production and use it for real-world applications

Open source

The source for this userguide is [available on Github](#) under the 3-clause [BSD license](#).

To build the userguide, install npm and run the following:

```
npm install  
npm run gitbook-dep  
npm run gitbook
```

The generated html will be located at `_book/index.html` .

Getting Started

In order to install GraphLab Create, please visit the [Installation page on turi.com](#) in order to

- register for a product key, and
- install the `graphlab-create` Python package.

Once installed, you will be able to read or run the example code and datasets by copying code into your terminal. Throughout the User Guide we assume you have imported the package via

```
import graphlab
```

Sometimes it is useful shorthand to use `gl` instead of `graphlab`. This can be done via

```
import graphlab as gl
```

Quick start

Here are a few commands to get started.

Task	code
Importing and parsing data	<pre>url = 'https://static.turi.com/datasets/millionsong/song_data.csv' songs = graphlab.SFrame.read_csv(url)</pre>
Visualizations	songs.show()
Computation on columns	songs['year'].mean()
Transforming columns	songs['num_words'] = songs['title'].apply(lambda x: len(x.split(' ')))
Aggregations	songs.groupby('artist_name', {'total': graphlab.aggregate.COUNT})
Create models	<pre>url = 'https://static.turi.com/datasets/regression/Housing.csv' x = graphlab.SFrame.read_csv(url) m = graphlab.linear_regression.create (x, target='price')</pre>

Working with Data

Data science often requires manipulating data so that it is ready for analysis. This section describes how to

- import data
- reshape unstructured data
- make transformations of existing data quickly
- compute summary statistics
- easily visualize your data

Working with Tabular Data

It's quite common that when you first get your hands on a dataset, it will be in a format that resembles a table. Tables are a straightforward format to use when cleaning data in preparation for more complicated data analysis, and the [SFrame](#) is the tabular data structure included with GraphLab Create. The SFrame is designed to scale to datasets much larger than will fit in memory.

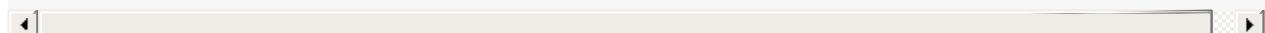
We will introduce the basics of the SFrame in the following chapters:

- [Loading and Saving](#) focuses on creating an SFrame from existing data in CSV format and how to persist an SFrame.
- The Frame supports a large number of common data manipulation operations and we will review a number of common ones in the chapter [Data Manipulation](#).
- [Apache Spark RDDs](#) goes into more detail about getting data in and out of Apache Spark RDDs.
- The chapter about [SQL databases](#) explains how to interface with relational data sources through Python DBAPI2 or ODBC.

Basic Loading and Saving

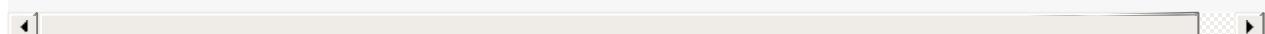
SFrames can import data in a [variety of formats](#), and we're always working on supporting more. A very common data format is the comma separated value (csv) file, which is what we'll use for these examples. We will use some data from the [Million Song Dataset](#) to aid our SFrame-related examples. This first table contains metadata about each song in the database. Here's how to load it into an SFrame:

```
songs = gl.SFrame.read_csv("https://static.turi.com/datasets/millionsong/song_data.csv")
```



Simple. No options are needed for the simplest case, as the SFrame parser infers column types. Of course, there are many options you may need to specify when importing a csv file. Some of the more common options come in to play when we load the usage data of users listening to these songs online:

```
usage_data = gl.SFrame.read_csv("https://static.turi.com/datasets/millionsong/10000.txt",
                                 header=False,
                                 delimiter='\t',
                                 column_type_hints={'X3':int})
```



The `header` and `delimiter` options are needed because this particular csv file does not provide column names in its first line, and the values are separated by tabs, not commas. The `column_type_hints` keeps the SFrame csv parser from attempting to infer the datatype of each column, which it does by default. For a full list of options when parsing csv files, check our [API Reference](#).

Once done we can inspect the first few rows of the tables we've imported.

```
songs
```

Columns:

song_id	str
title	str
release	str
artist_name	str
year	int

Rows: 1000000

song_id	title	artist_name	year
SOQMMHC12AB0180CB8	Silent Night		
SOVFVAK12A8C1350D9	Tanssi vaan		
SOGTUKN12AB017F4F1	No One Could Ever		
SOBNYVR12A8C13558C	Si Vos Quer\xc3\xa9s		
SOHSBXH12A8C13B0DF	Tangle Of Aspens		
SOZVAPQ12A8C13B63C	Symphony No. 1 G minor "Si ...		
SOQVRHI12A6D4FB2D7	We Have Got Love		
SOEYRFT12AB018936C	2 Da Beat Ch'yall		
SOPMIYT12A6D4F851E	Goodbye		
SOJCFMH12A8C13B0C2	Mama_ mama can't you see ?		
<hr/>			
release			
Monster Ballads X-Mas		Faster Pussy cat	2003
Karkuteill\xc3\xa4		Karkkiautomaatti	1995
Butter		Hudson Mohawke	2006
De Culo		Yerba Brava	2003
Rene Ablaze Presents Winte ...		Der Mystic	0
Berwald: Symphonies Nos. 1 ...		David Montgomery	0
Strictly The Best Vol. 34		Sasha / Turbulence	0
Da Bomb		Kris Kross	1993
Danny Boy		Joseph Locke	0
March to cadence with the ...	The Sun Harbor's Chorus-Do ...		0
...

[1000000 rows x 5 columns]

Note: Only the head of the SFrame is printed.

You can use `print_rows(num_rows=m, num_columns=n)` to print more rows and columns.

usage_data

```
Columns:
```

```
X1    str
X2    str
X3    int
```

```
Rows: 2000000
```

X1	X2	X3
b80344d063b5ccb3212f76538f ...	S0AKIMP12A8C130995	1
b80344d063b5ccb3212f76538f ...	S0BBMDR12A8C13253B	2
b80344d063b5ccb3212f76538f ...	S0BXHDL12A81C204C0	1
b80344d063b5ccb3212f76538f ...	S0BYHAJ12A6701BF1D	1
b80344d063b5ccb3212f76538f ...	S0DACBL12A8C13C273	1
b80344d063b5ccb3212f76538f ...	S0DDNQT12A6D4F5F7E	5
b80344d063b5ccb3212f76538f ...	S0DXRTY12AB0180F3B	1
b80344d063b5ccb3212f76538f ...	S0FGUAY12AB017B0A8	1
b80344d063b5ccb3212f76538f ...	S0FRQTD12A81C233C0	1
b80344d063b5ccb3212f76538f ...	S0HQWYZ12A6D4FA701	1
...

```
[2000000 rows x 3 columns]
```

Note: Only the head of the SFrame is printed.

You can use `print_rows(num_rows=m, num_columns=n)` to print more rows and columns.

Here we might want to rename columns from the default names:

```
usage_data.rename({'X1':'user_id', 'X2':'song_id', 'X3':'listen_count'})
```

SFrames can be saved as a csv file or in the SFrame binary format. If your SFrame is saved in binary format loading it is instantaneous, so we won't ever have to parse that file again. Here, the default is to save in binary format, and we supply the name of a directory to be created which will hold the binary files:

```
usage_data.save('./music_usage_data')
```

Loading is then very fast:

```
same_usage_data = gl.load_sframe('./music_usage_data')
```

In addition to these functions, JSON imports and exports, SQL/ODBC imports, and various Spark RDD conversion capabilities are also supported. For further information see the respective pages in the GraphLab Create API Documentation:

- [read_json](#)
- [export_json](#)
- [read_csv](#)
- [export_csv](#)
- [from_odbc](#)
- [from_odbc](#)
- [from_rdd](#)
- [to_rdd](#)
- [to_spark_dataframe](#)

For interfacing with Spark RDDs and relational databases see also the specific subsections in this user guide:

- [Spark RDDs](#)
- [SQL Databases](#)

Data Types

An SFrame is made up of columns of a contiguous type. For instance the `songs` SFrame is made up of 5 columns of the following types

```
song_id      str
title       str
release     str
artist_name   str
year        int
```

In this SFrame we see only string (`str`) and integer (`int`) columns, but a number of datatypes are supported:

- `int` (signed 64-bit integer)
- `float` (double-precision floating point)
- `str` (string)
- `array.array` (1-D array of doubles)
- `list` (arbitrarily list of elements)
- `dict` (arbitrary dictionary of elements)
- `datetime.datetime` (datetime with microsecond precision)
- `image` (image)

Data Manipulation

It isn't often that your dataset is "clean" enough to run one of our toolkits on it in a meaningful way right after import. Such is life...data is messy. SFrames enable you to complete data cleaning tasks in a scalable way, even on datasets that are much larger than your computer's memory.

Column Selection and Manipulation

A problem you may have noticed in the song metadata is that some songs' year value is 0. Suppose we want to change those to a missing value, so that they do not skew a summary statistic over the column (e.g. `mean` or `min`). If we knew about this before parsing, or are willing to parse the file again, we could add 0 to the `na_values` option of `read_csv`. Alternatively, we can apply an arbitrary function to one or multiple columns of an SFrame. Here's how to replace those zeroes with missing values with a Python lambda function:

```
songs['year'] = songs['year'].apply(lambda x: None if x == 0 else x)
songs.head(5)
```

song_id	title	release
SOQMMHC12AB0180CB8	Silent Night	Monster Ballads X-Mas
SOVFVAK12A8C1350D9	Tanssi vaan	Karkuteill\xc3\xa4
SOGTUKN12AB017F4F1	No One Could Ever	Butter
SOBNYVR12A8C13558C	Si Vos Quer\xc3\xa9s	De Culo
SOHSBXH12A8C13B0DF	Tangle Of Aspens	Rene Ablaze Presents Winte ...

artist_name	year
Faster Pussy cat	2003
Karkkiautomaatti	1995
Hudson Mohawke	2006
Yerba Brava	2003
Der Mystic	None

[5 rows x 5 columns]

Notice we had to reassign the resulting column back to our SFrame. This is because the content of the SFrame's columns (a separate data structure called an `SArray`, is immutable. SFrames can add and subtract columns liberally though, as it essentially is just a carrier of

references to SArrays.

We used a lambda function here because it is the simplest way to instantiate a small function like that. The `apply` method will take a named function (a normal Python function that starts with `def`) as well. As long as the function takes one parameter and returns one value, it can be applied to a column.

We can also apply a function to multiple columns. Suppose we want to add a column of the number of times the word 'love' is used in the `title` and `artist_name` column:

```
songs['love_count'] = songs[['title', 'artist_name']].apply(
    lambda row: sum(x.lower().split(' ').count('love') for x in row.values()))
songs.topk('love_count', k=5)
```

song_id	title	release	artist_name	year	love_count
SOMYDCX12A8AE4836B	The Love Story (Part 1) In ...	Skid Row / 34 Hours	Skid Row	None	4
SONXAVM12AB017AA1D	Document 15	Feels_ Feathers_ Bog and Bees	Low Low Low La La La Love ...	None	3
SOAWJOC12A8C1367A7	Black Black Window	Ends Of June	Low Low Low La La La Love ...	2007	3
SOXAVWF12A8AE4922C	One Piece	Low Low Low La La La Love ...	Low Low Low La La La Love ...	2007	3
SOAJRDR12A8C1383B1	Love Love Love	Radio Hitz	Spider Murphy Gang	None	3

[5 rows x 6 columns]

A few things to note here. We first select a subset of columns using a list within square brackets. This is useful in general, but it helps us with performance in this case, as fewer values are scanned. Also, when `apply` is called on an SFrame instead of an SArray as shown here, the input of the `lambda` function is a dictionary where the keys are the column names, and the values correspond to that row's values.

Another very useful and common operation is the ability to select columns based on types. For instance, this extracts all the columns containing strings.

```
song[str]
```

```
Data:
```

song_id	title
SOQMMHC12AB0180CB8	Silent Night
SOVFVAK12A8C1350D9	Tanssi vaan
SOGTUKN12AB017F4F1	No One Could Ever
SOBNYVR12A8C13558C	Si Vos Querés
SOHSBXH12A8C13B0DF	Tangle Of Aspens
SOZVAPQ12A8C13B63C	Symphony No. 1 G minor "Si..."
SOQVRHI12A6D4FB2D7	We Have Got Love
SOEYRFT12AB018936C	2 Da Beat Ch'yall
SOPMIYT12A6D4F851E	Goodbye
SOJCFMH12A8C13B0C2	Mama_ mama can't you see ?
release	artist_name
Monster Ballads X-Mas	Faster Pussy cat
Karkuteillä	Karkkiautomaatti
Butter	Hudson Mohawke
De Culo	Yerba Brava
Rene Ablaze Presents Winte...	Der Mystic
Berwald: Symphonies Nos. 1...	David Montgomery
Strictly The Best Vol. 34	Sasha / Turbulence
Da Bomb	Kris Kross
Danny Boy	Joseph Locke
March to cadence with the ...	The Sun Harbor's Chorus-Do...

Now, you may not feel comfortable transforming a column without inspecting more than the first 10 rows of it, as we did with the `year` column. To quickly get a summary of the column, we can do:

```
songs['year'].sketch_summary()
```

item	value	is exact
Length	1000000	Yes
Min	0.0	Yes
Max	2011.0	Yes
Mean	1030.325652	Yes
Sum	1030325652.0	Yes
Variance	997490.582407	Yes
Standard Deviation	998.744503067	Yes
# Missing Values	0	Yes
# unique values	90	No

Most frequent items:

value	count	2007	2006	2005	2008	2009	2004	2003	2002
0	484424	39414	37546	34960	34770	31051	29618	27389	23472
2001									
21604									

Quantiles:

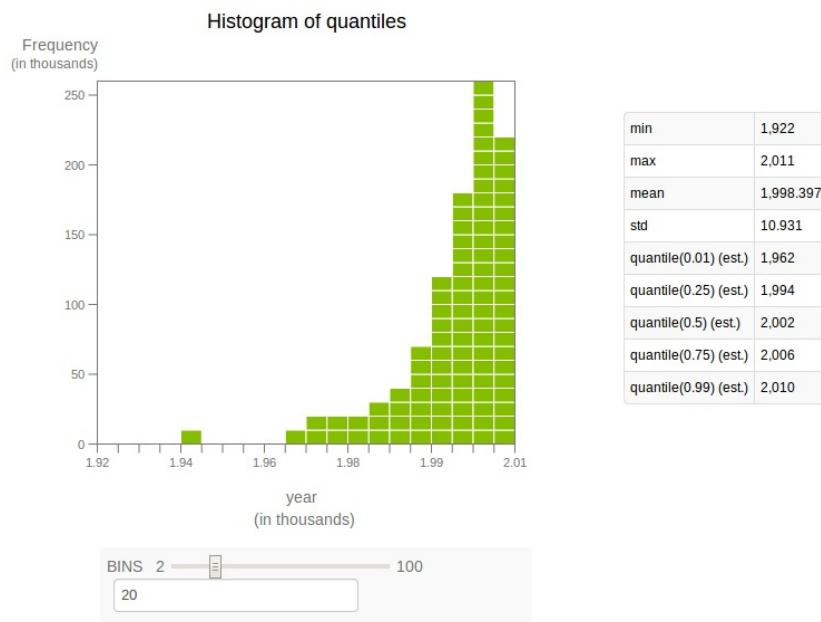
0%	1%	5%	25%	50%	75%	95%	99%	100%
0.0	0.0	0.0	0.0	1970.0	2002.0	2008.0	2009.0	2011.0

It appears that there are no other bogus years than 0. This summary splits its values into "exact" and "approximate". The approximate values could certainly be returned as exact values, but for the summary we use approximate values to make sure exploring large datasets is scalable. The methods we use only do a single pass of the data in the column, and each operation has well-defined bounds on how wrong the answer will be, which are listed in our [API Reference](#).

Using the most frequent items and quantiles described here, you can probably almost picture the distribution of years, where the tallest part is squarely within the 2000s. Fortunately, we don't have to just picture it in our heads. GraphLab Canvas provides visualizations of SFrames, as well as other data structures. GraphLab Canvas is covered in depth in the [Visualization](#) section. To view a histogram of these approximate quantiles, we run:

```
songs['year'].show()
```

Numeric Categorical



We have done some exploration, transformation, and feature generation. Let's spend some time filtering values we won't care about later. For example, perhaps we'll need an SFrame with **only** dated songs. This basic filter operation looks like this:

```
dated_songs = songs[songs['year'] != None]
dated_songs
```

```
+-----+-----+
|   song_id      |       title      |
+-----+-----+
| SOQMMHC12AB0180CB8 | Silent Night      |
| SOVFVAK12A8C1350D9 | Tanssi vaan      |
| SOGTUKN12AB017F4F1 | No One Could Ever |
| SOBNYVR12A8C13558C | Si Vos Quer\xc3\xaa9s |
| SOEYRFT12AB018936C | 2 Da Beat Ch'yall  |
| SOYGNWH12AB018191E | L'antarctique     |
| SOLJTLX12AB01890ED | El hijo del pueblo |
| SOMPVQB12A8C1379BB | Pilots           |
| SOSDCFG12AB0184647 | 006               |
| SOBARPM12A8C133DFF | (Looking For) The Heart Of ... |
+-----+
+-----+-----+-----+-----+
|       release      | artist_name      | year | love_count |
+-----+-----+-----+-----+
| Monster Ballads X-Mas | Faster Pussy cat | 2003 | 0          |
| Karkuteill\xc3\xaa4 | Karkkiautomaatti | 1995 | 0          |
| Butter                | Hudson Mohawke  | 2006 | 0          |
| De Culo                | Yerba Brava     | 2003 | 0          |
| Da Bomb                | Kris Kross       | 1993 | 0          |
| Des cobras des tarentules | 3 Gars Su'l Sofa | 2007 | 0          |
| 32 Grandes \xc3\x89xitos CD 2 | Jorge Negrete   | 1997 | 0          |
| The Loyal              | Tiger Lou        | 2005 | 0          |
| Lena 20 \xc3\x85r       | Lena Philipsson | 1998 | 0          |
| Cover Girl             | Shawn Colvin    | 1994 | 0          |
+-----+-----+-----+-----+
[? rows x 6 columns]
Note: Only the head of the SFrame is printed. This SFrame is lazily evaluated.
You can use len(sf) to force materialization.
```

The output does a good job at explaining what is happening here. SFrames will not do any work if it isn't required right away. This way, if you decide this filter operation isn't for you after looking at the first few rows, GraphLab Create won't waste computation time doing it anyways. However, it is important to verify that the missing values were indeed removed, and that we indeed removed 484424 rows, so we'll force materialization of the new SFrame.

```
len(dated_songs)
```

```
515576
```

Why does this filtering syntax work? What we're actually doing is placing an SArray in the square brackets. A comparison operator applied to an SArray returns a new SArray of the same length as the original, but with values that correspond to `true` or `false` based on the operator. Here's what it looks like:

```
songs['year'] != None
```

We may want to use more than one comparison operator, for which you must place each comparison statement in parentheses and use the bitwise boolean logic operators to combine the statements, as Python does not allow the overloading of logical operators. Perhaps we are building a music recommender with this data, and we would like to only use "reasonable" play counts, for some definition of reasonable:

```
reasonable_usage = usage_data[(usage_data['listen_count'] >= 10) & (usage_data['listen_count'] < 100)]
```

114026

You can also write a lambda function to filter using the `filter` function, which you can read about in the [API Reference](#).

Joins and Aggregation

Another important way to filter a dataset is to get rid of duplicate data. A nice way to search for duplicate data is to visualize the SFrame using GraphLab Canvas.

```
songs.show()
```

song_id	title	release	artist_name	year	love_count
dtype: str num_unique (est): 999,496	dtype: str num_unique (est): 700,812	dtype: str num_unique (est): 149,268	dtype: str num_unique (est): 72,848	dtype: int num_unique (est): 89	dtype: int num_unique (est): 5
num_undefined: 0	num_undefined: 0	num_undefined: 0	num_undefined: 0	num_undefined: 484,424	num_undefined: 0
frequent items:	frequent items:	frequent items:	frequent items:	min: 1,922	min: 0
No values appear with $\geq 0.01\%$ occurrence.	Intro Untitled Outro Interlude Home Silent Night Time Hold On Tonight Summertime Smile Introduction	Greatest Hits Live The Collection The Ultimate Collection The Very Best Of The Best Of The Platinum Collection Best Of Gold Anthology Original Album Classics Super Hits	Michael Jackson Johnny Cash Beastie Boys Joan Baez Aerosmith Jimi Hendrix Willie Nelson Kenny Rogers Neil Diamond Radiohead Franz Ferdinand The Doors	max: 2,011	max: 4
				mean: 1,998.397	mean: 0.03
				std: 10.931	std: 0.174
				distribution of values:	
					
				distribution of values:	
					

It appears our `song_id` is not completely unique. This would make merging the `songs` and `usage_data` SFrames error-prone if all we want to do is add song title information to the existing usage data. In this particular dataset, these repeat songs are included because they may have been released on several different albums (movie soundtracks, radio singles, etc.). If we do not care about which album release is included in the dataset, we can filter those duplicates like this:

```
other_cols = songs.column_names()
other_cols.remove('song_id')
agg_list = [gl.aggregate.SELECT_ONE(i) for i in other_cols]
unique_songs = songs.groupby('song_id', dict(zip(other_cols, agg_list)))
```

This code block needs some further explanation. It is centered around calling `groupby` with the `SELECT_ONE` aggregator. This selects a random representative row from each group. You must explicitly denote which columns will use this aggregator, so the list comprehension gathers all other columns than the one we are grouping by and uses the `SELECT_ONE` aggregator for each one. When used like this, `SELECT_ONE` will use the same random row for each column. This is great if it is not important which of the duplicates you pick. If it is, another aggregator like `MIN` or `MAX` may be in order.

Suppose we actually want to see the songs that have the highest play count? Now we can correctly group each song and aggregate its listen count, and then join the result to the song metadata to see the song titles.

```
tmp = usage_data.groupby(['song_id'], {'total_listens': gl.aggregate.SUM('listen_count'),
                                         'num_unique_users': gl.aggregate.COUNT('user_id')})
tmp.join(songs, ['song_id']).topk('total_listens')
```

song_id	num_unique_users	total_listens
SOBONKR12A58A7A7E0	6412	54136
SOAUWYT12A81C206F1	7032	49253
SOSXLTC12AF72A7F54	6145	41418
SOEGIYH12A6D4FC0E3	5385	31153
SOFRQTD12A81C233C0	8277	31036
SOAXGDH12A8C13F8A1	6949	26663
SONYKOW12AB01849C9	5841	22100
SOPUCYA12A8C13A694	3526	21019
SOUFTBI12AB0183F65	2887	19645
SOVDSJC12A58A7A271	2866	18309
title	release	
You're The One	If There Was A Way	
Undo	Vespertine Live	
Revelry	Only By The Night	
Horn Concerto No. 4 in E f ...	Mozart - Eine kleine Nacht ...	
Sehr kosmisch	Musik von Harmonia	
Dog Days Are Over (Radio Edit)	Now That's What I Call Mus ...	
Secrets	Waking Up	
Canada	The End Is Here	
Invalid	Fermi Paradox	
Ain't Misbehavin	Summertime	
artist_name	year	love_count
Dwight Yoakam	1990	0
Bj\ xc3\xb6rk	2001	0
Kings Of Leon	2008	0
Barry Tuckwell/Academy of ...	None	0
Harmonia	None	0
Florence + The Machine	None	0
OneRepublic	2009	0
Five Iron Frenzy	None	0
Tub Ring	2002	0
Sam Cooke	None	0

[10 rows x 8 columns]

The `usage_data` table is already in a great format for feeding to a recommender algorithm as it has user and song identifiers, and some form of a metric (number of listens) to rate how much the user liked the song. The problem here is that users that listened to a song a lot would skew the recommendations. At some point, once a user plays a song enough times, you know they really like it. Perhaps we could translate the `listen_count` into a rating instead. Here is a simple way to do it:

```
s = usage_data['listen_count'].sketch_summary()
import numpy
buckets = numpy.linspace(s.quantile(.005), s.quantile(.995), 5)
def bucketize(x):
    cur_bucket = 0
    for i in range(0,5):
        cur_bucket += 1
        if x <= buckets[i]:
            break
    return cur_bucket
usage_data['rating'] = usage_data['listen_count'].apply(bucketize)
usage_data
```

Columns:

user_id	str
song_id	str
listen_count	int
rating	int

Rows: 2000000

Data:

	user_id	song_id	listen_count	rating	
1	b80344d063b5ccb3212f76538f ...	SOAKIMP12A8C130995	1	1	
2	b80344d063b5ccb3212f76538f ...	S0BBMDR12A8C13253B	2	2	
3	b80344d063b5ccb3212f76538f ...	S0BXHDL12A81C204C0	1	1	
4	b80344d063b5ccb3212f76538f ...	S0BYHAJ12A6701BF1D	1	1	
5	b80344d063b5ccb3212f76538f ...	SODACBL12A8C13C273	1	1	
6	b80344d063b5ccb3212f76538f ...	SODDNQT12A6D4F5F7E	5	2	
7	b80344d063b5ccb3212f76538f ...	S0DXRTY12AB0180F3B	1	1	
8	b80344d063b5ccb3212f76538f ...	S0FGUAY12AB017B0A8	1	1	
9	b80344d063b5ccb3212f76538f ...	S0FRQTD12A81C233C0	1	1	
10	b80344d063b5ccb3212f76538f ...	S0HQWYZ12A6D4FA701	1	1	
	

[2000000 rows x 4 columns]

Note: Only the head of the SFrame is printed.

You can use `print_rows(num_rows=m, num_columns=n)` to print more rows and columns.

Working with Complex Types

SArrays are strongly-typed and some operations only work on certain types. Two types deserve some special consideration in this user guide: `list` and `dict`. These types can hold values of any type supported by SArrays, including themselves. So you can have an SFrame with a column of dictionaries that each have values that are lists of lists of dicts with mixed strings and integers... anyway, you get the idea!

The dataset we're working with right now does not have any of these types, so we can show how to convert one or several columns to one of these iterable types. For instance, suppose we want a list of all albums with a list of each song that was on the album. We would obtain this list from our metadata like this:

```
albums = songs.groupby(['release','artist_name'], {'tracks': gl.aggregate.CONCAT('title'),
                                                    'years': gl.aggregate.CONCAT('year')})
albums
```

	artist_name	release	
	Veruca Salt	Eight Arms To Hold You	
	Les Compagnons De La Chanson	Heritage - Le Chant De Mal ...	
	Nelly / Fat Joe / Young Tr ...	Sweat	
	The Grouch	My Baddest B*tches	
	Ozzy Osbourne	Diary of a madman / Bark a ...	
	Peter Hunnigale	Reggae Hits Vol. 32	
	Burning Spear	Studio One Classics	
	Bond	New Classix 2008	
	Lee Coombs feat. Katherine ...	Control	
	Stevie Wonder	Songs In The Key Of Life	

	tracks	years	
	['One Last Time', 'With ...']	array('d', [1997.0, 1997.0 ...])	
	['I Commedianti', 'Il Est ...']	None	
	['Grand Hang Out']	array('d', [2004.0])	
	['Silly Putty (Zion I Feat ...']	array('d', [1999.0])	
	["You Can't Kill Rock And ...']	array('d', [1981.0, 1983.0 ...])	
	['Weeks Go By']	None	
	['Rocking Time']	array('d', [1973.0])	
	['Allegretto', 'Kashmir']	None	
	['Control (10Rapid Remix)' ...']	None	
	['Ngiculela-Es Una Histori ...']	array('d', [1976.0])	
	

[230558 rows x 4 columns]

Note: Only the head of the SFrame is printed.

You can use `print_rows(num_rows=m, num_columns=n)` to print more rows and columns.

The [CONCAT](#) aggregator simply creates a list of all values in the given column for each group. I included the year for debugging purposes, since I didn't know if the invariant of "every song on the same release has the same year" was true in this dataset. Even looking at the first row, this is clearly not true:

```
albums[0]
```

```
{'artist_name': 'Veruca Salt',
'release': 'Eight Arms To Hold You',
'tracks': ['With David Bowie',
'One Last Time',
'The Morning Sad',
'Awesome',
'Stoneface',
"Don't Make Me Prove It",
'Straight',
'Earthcrosser',
'Benjamin',
'Volcano Girls',
'Shutterbug',
'Sound Of The Bell',
'Loneliness Is Worse',
'Venus Man Trap'],
'years': array('d', [1997.0, 1997.0, 1997.0, 1997.0, 1997.0, 1997.0, 1994.0, 1997.0, 19
```

In light of this complication, and for demonstration purposes, let's have our 'tracks' column contain dictionaries instead, where the key is the year and the value is a list of tracks.

```
albums = songs.groupby(['release','artist_name','year'], {'tracks':gl.aggregate.CONCAT('t
albums = albums.unstack(column=['year','tracks'], new_column_name='track_dict')
albums
```

```
+-----+-----+
|      artist_name      |      release      |
+-----+-----+
| Veruca Salt          | Eight Arms To Hold You   |
| Les Compagnons De La Chanson | Heritage - Le Chant De Mal ... |
| Nelly / Fat Joe / Young Tr ... | Sweat           |
| The Grouch          | My Baddest B*tches    |
| Ozzy Osbourne        | Diary of a madman / Bark a ... |
| Peter Hunnigale       | Reggae Hits Vol. 32     |
| Burning Spear        | Studio One Classics   |
| Bond                 | New Classix 2008       |
| Lee Coombs feat. Katherine ... | Control          |
| Stevie Wonder         | Songs In The Key Of Life |
+-----+-----+
+-----+
|      track_dict      |
+-----+
| {1994: [\'Straight\"], 199 ... |
|     None               |
| {2004: ['Grand Hang Out']} |
| {1999: ['Simple Man']}  |
| {1986: [\'Secret Loser\"], ... |
|     None               |
| {1973: ['Rocking Time']} |
|     None               |
|     None               |
| {1976: ['Have A Talk With ... |
|     ...                |
+-----+
[230558 rows x 3 columns]
Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.
```

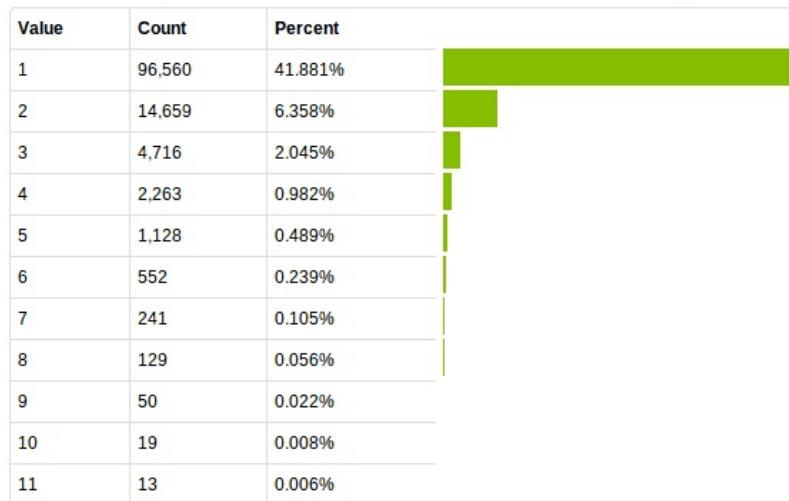
The `unstack` function is essentially a convenience function for calling `groupby` on all other columns with the `CONCAT` aggregator. This operation can be undone with `stack`, which I won't take time to show in detail.

With our data organized this way, we can get some statistics on how many albums have this oddity and see that it's actually somewhat prevalent for this to occur:

```
albums['num_years'] = albums['track_dict'].item_length()
albums['num_years'].show()
```

Numeric Categorical

Most frequent items from <temporary SArrayView>



We can still recover the full track listing with some dict operations, so suppose we want both a `list` and a `dict` representation of the tracks.

```
albums['track_list'] = albums['track_dict'].dict_values()
albums
```

```
+-----+-----+
|      artist_name      |      release      |
+-----+-----+
| Veruca Salt          | Eight Arms To Hold You   |
| Les Compagnons De La Chanson | Heritage - Le Chant De Mal ... |
| Nelly / Fat Joe / Young Tr ... | Sweat           |
| The Grouch          | My Baddest B*tches    |
| Ozzy Osbourne        | Diary of a madman / Bark a ... |
| Peter Hunnigale       | Reggae Hits Vol. 32      |
| Burning Spear         | Studio One Classics   |
| Bond                 | New Classix 2008       |
| Lee Coombs feat. Katherine ... | Control          |
| Stevie Wonder         | Songs In The Key Of Life |
+-----+-----+
+-----+-----+
|      track_dict      | num_years |      track_list      |
+-----+-----+
| {1994: ['Straight'], 199 ... | 2 | [['Straight'], ['With D ... |
| None                  | None | None           |
| {2004: ['Grand Hang Out']} | 1 | [['Grand Hang Out']] |
| {1999: ['Simple Man']} | 1 | [['Simple Man']] |
| {1986: ['Secret Loser'], ... | 3 | [["You Can't Kill Rock An ... |
| None                  | None | None           |
| {1973: ['Rocking Time']} | 1 | [['Rocking Time']] |
| None                  | None | None           |
| None                  | None | None           |
| {1976: ['Have A Talk With ... | 1 | [['Have A Talk With God']] |
| ...                   | ... | ...           |
+-----+-----+
[230558 rows x 5 columns]
Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.
```

We have almost recovered our old column, but we have a list of lists instead of just a single list. Fixing that is a simple `apply` away:

```
import itertools
albums['track_list'] = albums['track_list'].apply(lambda x: list(itertools.chain(*x)))
albums
```

```
+-----+-----+
|      artist_name      |      release      |
+-----+-----+
| Veruca Salt          | Eight Arms To Hold You   |
| Les Compagnons De La Chanson | Heritage - Le Chant De Mal ... |
| Nelly / Fat Joe / Young Tr ... | Sweat           |
| The Grouch          | My Baddest B*tches    |
| Ozzy Osbourne        | Diary of a madman / Bark a ... |
| Peter Hunnigale       | Reggae Hits Vol. 32      |
| Burning Spear         | Studio One Classics   |
| Bond                 | New Classix 2008       |
| Lee Coombs feat. Katherine ... | Control          |
| Stevie Wonder         | Songs In The Key Of Life |
+-----+-----+
+-----+-----+
|      track_dict      | num_years |      track_list      |
+-----+-----+
| {1994: [\'Straight\"], 199 ... | 2 | [\'Straight\", \'With Davi ... |
| None                  | None | None           |
| {2004: ['Grand Hang Out']} | 1 | ['Grand Hang Out'] |
| {1999: ['Simple Man']} | 1 | ['Simple Man'] |
| {1986: [\'Secret Loser\"], ... | 3 | ["You Can\'t Kill Rock And ... |
| None                  | None | None           |
| {1973: ['Rocking Time']} | 1 | ['Rocking Time'] |
| None                  | None | None           |
| None                  | None | None           |
| {1976: ['Have A Talk With ... | 1 | ['Have A Talk With God'] |
| ...                   | ... | ...           |
+-----+-----+
[230558 rows x 5 columns]
Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.
```

We can also filter elements from within a list or dictionary. Here is how to remove all songs made in 1994 or 1999:

```
albums['track_dict'] = albums['track_dict'].dict_trim_by_keys([1994, 1999])
albums
```

```
+-----+-----+
|      artist_name      |      release      |
+-----+-----+
| Veruca Salt          | Eight Arms To Hold You   |
| Les Compagnons De La Chanson | Heritage - Le Chant De Mal ... |
| Nelly / Fat Joe / Young Tr ... | Sweat           |
| The Grouch          | My Baddest B*tches    |
| Ozzy Osbourne        | Diary of a madman / Bark a ... |
| Peter Hunnigale       | Reggae Hits Vol. 32      |
| Burning Spear         | Studio One Classics   |
| Bond                 | New Classix 2008       |
| Lee Coombs feat. Katherine ... | Control          |
| Stevie Wonder         | Songs In The Key Of Life |
+-----+-----+
+-----+-----+
|      track_dict      | num_years |      track_list      |
+-----+-----+
| {1997: [\'With David Bowie ... | 2 | [\'Straight\', \'With Davi ... |
|     None               | None | None                   |
| {2004: ['Grand Hang Out']} | 1 | ['Grand Hang Out']    |
|     {}                | 1 | ['Simple Man']        |
| {1986: [\'Secret Loser\'], ... | 3 | ["You Can\'t Kill Rock And ... |
|     None               | None | None                   |
| {1973: ['Rocking Time']} | 1 | ['Rocking Time']       |
|     None               | None | None                   |
|     None               | None | None                   |
| {1976: ['Have A Talk With ... | 1 | ['Have A Talk With God'] |
|     ...                | ... | ...                   |
+-----+-----+
[230558 rows x 5 columns]
Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.
```

Since dictionaries are good for item lookups, we can filter by elements we find in dictionaries. Here is a query that filters albums by whether any of their songs came out in 1965:

```
albums[albums['track_dict'].dict_has_any_keys(1965)]
```

```
+-----+-----+
|      artist_name      |      release      |
+-----+-----+
| Jr. Walker & The All Stars | The Motown Story: The Sixties |
|      The Seekers        | The Best Sixties Album In ... |
| Jr. Walker & The All Stars | Sparks Present Motown Made ... |
|      The Who            | Who's Next          |
|      Bobby Vinton       | The Best Of Bobby Vinton |
| Righteous Brothers     | The Collection      |
|      Little Milton      | Chess Blues         |
|      The Who            | Singles Box         |
|      Dionne Warwick     | Here I Am          |
|      Albert Ayler        | Nuits De La Fondation Maeg ... |
+-----+-----+
+-----+-----+
|      track_dict      | num_years |      track_list      |
+-----+-----+
| {1965: ['Shotgun']} | 1 |      ['Shotgun'] |
| {1965: ['The Carnival Is O ... | 1 |      ['The Carnival Is Over'] |
| {1965: ["Cleo\\'s Back"]} | 1 |      ["Cleo\\'s Back"] |
| {1995: ["I Don\\'t Even Kno ... | 4 |      [\\'My Generation\\', \\'Behi ... |
| {1991: [\\'Please Love Me F ... | 6 |      [\\'Roses Are Red (My Love) ... |
| {2006: ['Let The Good Time ... | 4 |      ['Justine', 'Stagger Lee', ... |
| {1965: ["We\\'re Gonna Make ... | 1 |      ["We\\'re Gonna Make It"] |
| {1965: ['Shout And Shimmy']} | 1 |      ['Shout And Shimmy'] |
| {1965: ['How Can I Hurt Yo ... | 1 |      ['How Can I Hurt You? (LP ... |
| {1969: ['Music Is The Heal ... | 4 |      ['Spirits', 'Spirits Rejoi ... |
+-----+-----+
[? rows x 5 columns]
Note: Only the head of the SFrame is printed. This SFrame is lazily evaluated.
You can use len(sf) to force materialization.
```

Converting to a `list` or `dict` doesn't need to group by the rest of the values in the row. If for some reason you want to turn this into a table with one column and all values packed into a list, you can!

```
big_list = albums.pack_columns(albums.column_names())
big_list
```

```
+-----+
|          X1          |
+-----+
| ['Veruca Salt', 'Eight ... |
| ['Les Compagnons De La Cha ... |
| ['Nelly / Fat Joe / Young ... |
| ['The Grouch', 'My Baddest ... |
| ['Ozzy Osbourne', 'Diar ... |
| ['Peter Hunnigale', 'Regga ... |
| ['Burning Spear', 'Studio ... |
| ['Bond', 'New Classix 2008 ... |
| ['Lee Coombs feat. Katheri ... |
| ['Stevie Wonder', 'Songs I ... |
| ...
+-----+
[230558 rows x 1 columns]
Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.
```

The `unpack` function accomplishes the reverse task. These examples may be a bit contrived, but these functions are very useful when working with unstructured data like text, as you will be able to see in the [Text Analysis](#) chapter of this guide.

To find out more, check out the [API Reference for SFrames](#) and the [hands-on exercises](#) at the end of the chapter.

Spark Integration

GraphLab Create has the ability to convert [Apache Spark's Resilient Distributed Datasets \(RDD\)](#) to an SFrame and back.

Setup the Environment

To use GraphLab Create within PySpark, you need to set the `$SPARK_HOME` and `$PYTHONPATH` environment variables on the driver. A common usage:

```
export PYTHONPATH=$SPARK_HOME/python/:$SPARK_HOME/python/lib/py4j-0.8.2.1-src.zip:$PYTHONPATH  
export SPARK_HOME =
```

Run from the PySpark Python Shell

```
cd $SPARK_HOME  
bin/pyspark
```

Run from a standard Python Shell

Make sure you have exported the `PYTHONPATH` and `SPARK_HOME` environment variables. Then run (for example):

```
ipython
```

Then you need to start spark:

```
from pyspark import SparkContext  
from pyspark.sql import SQLContext  
# Launch spark by creating a spark context  
sc = SparkContext()  
# Create a SparkSQL context to manage dataframe schema information.  
sql = SQLContext(sc)
```

Make an SFrame from an RDD

```
from graphlab import SFrame
rdd = sc.parallelize([(x,str(x), "hello") for x in range(0,5)])
sframe = SFrame.from_rdd(rdd, sc)
print sframe
```

```
+-----+
|      X1      |
+-----+
| [0, 0, hello] |
| [1, 1, hello] |
| [2, 2, hello] |
| [3, 3, hello] |
| [4, 4, hello] |
+-----+
[5 rows x 1 columns]
```

Make an SFrame from a Dataframe (preferred)

```
from graphlab import SFrame
rdd = sc.parallelize([(x,str(x), "hello") for x in range(0,5)])
df = sql.createDataFrame(rdd)
sframe = SFrame.from_rdd(df, sc)
print sframe
```

```
+----+----+----+
| _1 | _2 | _3 |
+----+----+----+
| 0 | 0 | hello |
| 1 | 1 | hello |
| 2 | 2 | hello |
| 3 | 3 | hello |
| 4 | 4 | hello |
+----+----+----+
[5 rows x 3 columns]
```

Make an RDD from an SFrame

```
from graphlab import SFrame
sf = gl.SFrame({'x': [1,2,3], 'y': ['fish', 'chips', 'salad']})
rdd = sf.to_rdd(sc)
rdd.collect()
```

```
[0, '0', 'hello'),
(1, '1', 'hello'),
(2, '2', 'hello'),
(3, '3', 'hello'),
(4, '4', 'hello')]
```

Make a DataFrame from an SFrame (preferred)

```
from graphlab import SFrame
sf = gl.SFrame({'x': [1,2,3], 'y': ['fish', 'chips', 'salad']})
df = sf.to_spark_dataframe(sc,sql)
df.show()
```

```
+---+---+
| x| y|
+---+---+
| 1| fish|
| 2| chips|
| 3| salad|
+---+---+
```

Requirements and Caveats

- The currently release requires Python 2.7, Spark 1.3 or later, and the `hadoop` binary must be within the `PATH` of the driver when running on a cluster or interacting with Hadoop (e.g., you should be able to run `hadoop classpath`).
- We also currently only support Mac and Linux platforms but will have Windows support soon.
- The GraphLab integration with Spark supports Spark execution modes `local`, `yarn-client`, and `standalone` `spark://<hostname:port>`. ("yarn-cluster" is not available through PySpark)

Recommended Settings for Spark Installation on a Cluster

We recommend downloading `Pre-built for Hadoop 2.4 and later` version of [Apache Spark](#).

Notes

1. RDD conversion works with GraphLab Create **right out of the box**. No additional Spark setup is required. When you install GraphLab Create, it comes with a JAR that enables this feature. To find the location of the JAR file, execute this command:

```
graphlab.get_spark_integration_jar_path()
```

2. GraphLab Create can only convert to types it supports. This means that if you have an RDD with Python types other than int, long, str, list, dict, array.array, or datetime.datetime (image is not supported for conversion currently), your conversion may fail (when using Spark locally, you may get lucky and successfully convert an unsupported type, but it will probably fail on a YARN cluster).
3. SFrames fit most naturally with DataFrame. Both have strict column types and they have a similar approach to storing data. This is why we also have a `graphlab.SFrame.to_spark_dataframe` method. The `graphlab.SFrame.from_rdd` method works with both DataFrame and any other rdd, so there is no `from_dataframe` method.

Introduction

There are two ways to read data from a SQL database in to GraphLab Create:

- [Python DBAPI2](#)
- [ODBC](#)

DBAPI2 support is a new feature and currently released as beta, but we strongly encourage you to try it first. The ease of getting started with DBAPI2 far surpasses using ODBC.

DBAPI2 Integration

[DBAPI2](#) is a standard written to encourage database providers to expose a common interface for executing SQL queries when making Python modules for their database. Common usage of a DBAPI2-compliant module from Python looks something like this:

```
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()

# Create table
c.execute('''CREATE TABLE stocks
             (date text, trans text, symbol text, qty real, price real)''')

# Insert a row of data
c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")

# Save (commit) the changes
conn.commit()

c.execute("SELECT * FROM stocks")
results = c.fetchall()
```

(example adapted from [here](#))

SFrame offers a DBAPI2 integration that enables you to read and write SQL data in a similar, concise fashion. Using the connection object in the previous example, here is how you would read the data as an SFrame using the `from_sql` method:

```
import graphlab as gl
stocks_sf = gl.SFrame.from_sql(conn, "SELECT * FROM stocks")
```

If you would like to then write this table to the database, that's easy too, using the `to_sql` method. `to_sql` simply attempts to append to an already existing table, so if you intend to write the data to a new table in your database, then you must use the "CREATE TABLE" syntax, including the type syntax supported by your database. Here's an example of creating a new table and then appending more data to the table.

```
import datetime as dt
c = conn.cursor()

c.execute('''CREATE TABLE more_stocks
            (date text, trans text, symbol text, qty real, price real)''')
c.commit()
stocks_sf.to_sql(conn, "more_stocks")

# Append another row
another_row = gl.SFrame({'date':[dt.datetime(2006, 3, 28)],
                         'trans':['BUY'],
                         'symbol':['IBM'],
                         'qty':[1000],
                         'price':[45.00]})

another_row.to_sql(conn, "more_stocks")
```

That is all there is to know to get started using SFrames with Python DBAPI2 modules! For more details you can consult the API documentation of `from_sql` and `to_sql`. Currently, we have tested our DBAPI2 support with these modules:

- MySQLdb
- psycopg2
- sqlite3

This means that our DBAPI2 support may or may not work on other modules claiming to be DBAPI2-compliant. We will be adding more modules to this list as driven by what our users are interested in, so if you are interested in other modules, please try them out and let us know! If there is an issue with using one, please file an issue on [our GitHub page](#) and include the error output you received and/or some small code sample that exhibits the error. You can even [submit a pull request](#) if you are able to fix the issue.

If your database does not support a DBAPI2 python module, but does support an ODBC driver, keep reading.

ODBC Integration

[ODBC](#) stands for "Open Database Connectivity". It is an old standard (first version was released in 1992) that provides a language-agnostic interface for programs to access data in SQL databases. There are a few extra steps to set it up and extra concepts to learn before you start using it, but it remains one of the most universal ways to communicate with SQL databases. The ODBC connector included in SFrame only supports Linux and OS X. Windows is not supported at this time.

ODBC Overview

ODBC provides maximum portability by requiring the database vendor to write a driver that implements a common SQL-based interface. One or more of these drivers are managed by a system-wide ODBC driver manager. This means that in order to use ODBC, you must first install an ODBC driver manager, and then find your database's ODBC driver, download it, and install it into the driver manager. The database itself need not be installed on your computer; it can be installed on a remote machine. It is very important to make sure your ODBC driver works with your database before trying GraphLab Create's ODBC functions to make sure you are debugging the correct problem. The next section will help you do this.

Setting Up An ODBC Environment

The only ODBC driver manager we officially support is [unixODBC](#). You are welcome to try others if you really want to, but we do not guarantee that this will work. If you are so bold, let us know what happened!

Execute this command to install unixODBC on Ubuntu:

```
sudo apt-get install unixodbc
```

this on CentOS 6:

```
sudo yum install unixODBC.x86_64
```

and this on OS X (if you use Homebrew):

```
brew install unixodbc
```

Once you have this installed, try executing this command:

```
odbcinst -j
```

```
[SQLite]
Description=SQLite ODBC Driver
; Replace with your own path
Driver=/path/to/lib/libsqliteodbc.so
Setup=/path/to/lib/libsqLiteodbc.so
UsageCount=1

[myodbc]
Description = mySQL ODBC driver
Driver = /path/to/lib/libmyodbc.so
Setup = /path/to/lib/libodbcmyS.so
Debug = 0
CommLog = 1
UsageCount = 1
```

Setting Up A Data Source

ODBC has the concept of a "data source" (or DSN for "data source name"), which corresponds to a specific database. For example, if you have mySQL installed on your system, you'll need to create a data source to point to a specific database within that system. To do this, you must add an entry to the file that is responsible for either "SYSTEM DATA SOURCES" or "USER DATA SOURCES" from the output of your "odbcinst -j" command. Here is an example of how you could set up a SQLite DSN, adapted from [here](#):

```
[sqlite_dsn_name]
Description=My SQLite test database
; corresponds to above driver installation entry
Driver=SQLite
Database=/home/johndoe/databases/mytest.db
; optional lock timeout in milliseconds
Timeout=2000
```

and an example for mySQL, adapted from [here](#):

```
[mysql_dsn_name]
Description=myodbc
; Assumes your driver is installed with the name "myodbc"
Driver=myodbc
; Name of the database you want to connect to within mySQL
Database=test
Server=localhost
Port=3306
```

It's a great idea to test that all of this works before unleashing GraphLab Create on your database. UnixODBC comes with a command line utility called isql that will access your database through ODBC. This is not a very full-featured command line tool, so we only

recommend using it for testing if your ODBC setup works. Invoke it like this to access our example mySQL DSN:

```
isql mysql_dsn_name username password
```

If you are able to do something simple to your database, then feel free to move on to GraphLab Create's ODBC functions. Just so you don't have to use your brain, here's what your isql output should roughly look like when you do something simple:

```
$ isql mysql_dsn_name myusername mypassword
+-----+
| Connected! |
|           |
| sql-statement |
| help [tablename] |
| quit          |
|
+-----+
SQL> CREATE TABLE foo (a INTEGER, b INTEGER)
SQLRowCount returns 1
SQL> INSERT INTO foo VALUES(1, 2)
SQLRowCount returns 1
SQL> SELECT * FROM foo
+-----+
| a      | b      |
+-----+-----+
| 1      | 2      |
+-----+
SQLRowCount returns 0
1 rows fetched
```

If you aren't able to do something like the above, make sure to read the documentation of your specific ODBC driver to see if you missed any part of setup. Since there are so many drivers, we can't possibly test them all and document their many intricacies.

Example: Step-by-Step Instructions for MySQL on OSX

1. Install the ODBC Driver Manager, unixodbc

```
brew install unixodbc
```

Note: If you do not have Homebrew installed on OSX, see installation instructions [here](#).

2. Confirm ODBC Driver Manager Installation and Configuration Settings

```
odbcinst -j
```

Sample output:

```
rajat@fourier ~> odbcinst -j
unixODBC 2.3.2
DRIVERS.....: /usr/local/Cellar/unixodbc/2.3.2_1/etc/odbcinst.ini
SYSTEM DATA SOURCES: /usr/local/Cellar/unixodbc/2.3.2_1/etc/odbc.ini
FILE DATA SOURCES..: /usr/local/Cellar/unixodbc/2.3.2_1/etc/ODBCDataSources
USER DATA SOURCES..: /Users/rajat/.odbc.ini
SQLULEN Size.....: 8
SQLLEN Size.....: 8
SQLSETPOSIROW Size.: 8
```

This command will also let you know where the .ini files for the drivers and data sources need to be created.

From this, the Drivers go here: `/usr/local/Cellar/unixodbc/2.3.2_1/etc/odbcinst.ini` and User Data Sources (DSN definitions for databases) go here: `/Users/rajat/.odbc.ini`.

3. Install MySQL ODBC Driver for Mac

```
brew install mysql-connector-odbc
```

This will install in `/usr/local/Cellar/mysql-connector-odbc`

4. Find Installed MySQL Driver

We need to find the .so file for the actual driver (so it can be registered with the ODBC Driver Manager), for this installation it is here:

```
rajat@fourier ~> ll /usr/local/Cellar/mysql-connector-odbc/5.3.2_1/lib
total 14152
-r--r--r-- 1 rajat admin 3623032 Dec 18 12:14 libmyodbc5a.so
-r--r--r-- 1 rajat admin 3619008 Dec 18 12:14 libmyodbc5w.so
```

We want `libmyodbc5w.so` so we can support Unicode.

Now that we know this, we need to register this driver with the ODBC Driver Manager by manually creating the .ini file for the driver in the location we learned earlier from `odbcinst -j`.

5. Register driver with ODBC Driver Manager

Manually create the entry for the driver in the DRIVERS .ini mentioned from `odbcinst -j` , here is what it should look like:

```
rajat@fourier ~> cat /usr/local/Cellar/unixodbc/2.3.2_1/etc/odbcinst.ini
```

```
[myodbc]
Description = MySQL ODBC Driver
Driver = /usr/local/Cellar/mysql-connector-odbc/5.3.2_1/lib/libmyodbc5w.so
Setup = /usr/local/Cellar/mysql-connector-odbc/5.3.2_1/lib/libmyodbc5w.so
Debug = 0
CommLog = 1
UsageCount = 1
```

6. Create the Database definition as a DSN

Now we need to create the DSN definition in the USER DATA SOURCES location returned by `odbcinst -j` , from this output we need to edit `/Users/rajat/.odbc.ini` . Notice that the Driver field below refers to the name of the section added in the previous step (myodbc).

Remember to update the Database, Server, and Port fields appropriately for your machine.

```
[mysqltest]
Description=myodbc
Driver=myodbc
Database=test
Server=localhost
Port=3306
```

7. Done! Test from GraphLab Create

```
import graphlab
conn = graphlab.connect_odbc('DSN=mysqltest;UID=root;PWD=foo')
print conn.dbms_name
'MySQL'
```

Other Resources

Here are some posts that we found to be helpful when testing specific database drivers. Don't follow them blindly, but take them in context. This is just to save you a bit of googling if you're stuck setting up your ODBC environment. When in doubt, always rely on the driver's official documentation.

- [SAP HANA DB](#)
- [PostgreSQL](#)

- Microsoft SQL Server
- SQLite
- Various drivers
- Help with forming connection strings

Using ODBC Within GraphLab Create

Adding a DSN makes forming your connection string much easier. An ODBC connection string is similar to the database connection strings you may be familiar with, but slightly different. For our running mySQL example, this would be the connection string:

```
'DSN=mysql_dsn_name;UID=myusername;PWD=mypassword'
```

Therefore, to connect to this database through GraphLab Create, you would execute this in Python:

```
import graphlab as gl  
db = gl.connect_odbc('DSN=mysql_dsn_name;UID=myusername;PWD=mypassword')
```

As long as you did not receive an error message in that last step, you can read the result of any SQL query like so:

```
sf = gl.SFrame.from_odbc(db, "SELECT * FROM foo")
```

Now feel free to use your SFrame as you please! If you would like to write an SFrame to a table in your database, we support creating a new table, and appending to an existing table. Both can be achieved through the same function call:

```
sf.to_odbc(db, 'a_table_name')
```

If the table name is found to exist in your database, `to_odbc` will attempt to append each row to the table it finds. There is nothing sophisticated about this, as `to_odbc` does not do any sort of type checking or column matching in this case.

If the table name is not found, `to_odbc` will use a heuristic to pick the best type specific to your database for each column of the SFrame and create the table.

If you find yourself needing to execute arbitrary SQL commands to prepare your environment for a query (and which may not return results), you can call `execute_query` on your database connection object:

```
db.execute_query("SET SCHEMA foo_schema")
```

Notes

We do not support writing all types that are possible to hold in an SFrame, namely list, dict, or image types. This is because there is no clean mapping to an ODBC type.

We support reading all [ODBC types](#) except time intervals. Reading SQL time intervals may work for certain drivers, but your mileage may vary so we are not officially supporting it at this time. Also, SFrames do not support timestamps that use fractions of seconds, so the fraction portion of a timestamp will be ignored when reading.

Working with Graphs

Graphs allow us to understand complex networks by focusing on relationships between pairs of items. Each item is represented by a *vertex* in the graph, and relationships between items are represented by *edges*.

To facilitate graph-oriented data analysis, GraphLab Create offers a `SGraph` object, a scalable graph data structure backed by SFrames. In this chapter, we show that SGraphs allow arbitrary dictionary attributes on vertices and edges, flexible vertex and edge query functions, and seamless transformation to and from SFrames.

Creating an SGraph

There are several ways to create an SGraph. The simplest is to start with an empty graph, then add vertices and edges in the form of lists of `graphlab.Vertex` and `graphlab.Edge` objects. SGraphs are structually immutable; in the following snippet, `add_vertices` and `add_edges` both return a new graph.

```
from graphlab import SGraph, Vertex, Edge
g = SGraph()
verts = [Vertex(0, attr={'breed': 'labrador'}),  

         Vertex(1, attr={'breed': 'labrador'}),  

         Vertex(2, attr={'breed': 'vizsla'})]
g = g.add_vertices(verts)
g = g.add_edges(Edge(1, 2))
print g
```

```
SGraph({'num_edges': 1, 'num_vertices': 3})  
Vertex Fields:[ '__id', 'breed' ]  
Edge Fields:[ '__src_id', '__dst_id' ]
```

We can chain these steps together to make a new graph in a single line.

```
g = SGraph().add_vertices([Vertex(i) for i in range(10)]).add_edges(  
    [Edge(i, i+1) for i in range(9)])
```

SGraphs can also be created from an edge list stored in an SFrame. Vertices are added to the graph automatically based on the edge list, and columns of the SFrame not used as source or destination vertex IDs are assumed to be edge attributes. For this example we download a dataset of James Bond characters to an SFrame, then build the graph.

```
from graphlab import SFrame
edge_data = SFrame.read_csv(
    'https://static.turi.com/datasets/bond/bond_edges.csv')

g = SGraph()
g = g.add_edges(edge_data, src_field='src', dst_field='dst')
print g
```

```
SGraph({'num_edges': 20, 'num_vertices': 10})
```

The SGraph constructor also accepts vertex and edge SFrames directly. We can construct the same James Bond graph with the following two lines:

```
vertex_data = SFrame.read_csv('https://static.turi.com/datasets/bond/bond_vertices.csv')

g = SGraph(vertices=vertex_data, edges=edge_data, vid_field='name',
           src_field='src', dst_field='dst')
```

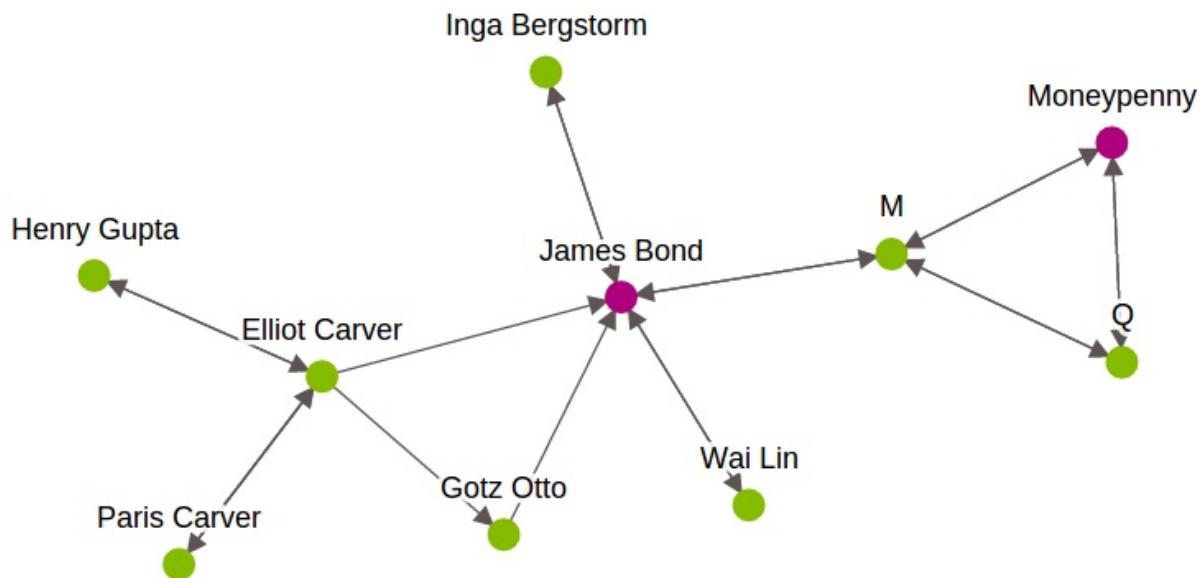
Finally, an SGraph can be created directly from a file, either local or remote, using the `graphlab.load_sgraph()` method. Loading a graph with this method works with both the native binary save format and a variety of text formats. In the following example we save the SGraph in binary format to a new folder called "james_bond", then re-load it under a different name.

```
g.save('james_bond')
new_graph = graphlab.load_sgraph('james_bond')
```

Inspecting SGraphs

Small graphs can be explored very efficiently with the `SGraph.show` method, which displays a plot of the graph. The vertex labels can be IDs or any vertex attribute.

```
g.show(vlabel='id', highlight=['James Bond', 'Moneypenny'], arrows=True)
```



For large graphs visual displays are difficult, but graph exploration can still be done with the `sgraph.summary` ---which prints the number of vertices and edges---or by retrieving and plotting subsets of edges and vertices.

```
print g.summary()
```

```
{'num_edges': 20, 'num_vertices': 10}
```

To retrieve the contents of an SGraph, the `get_vertices` and `get_edges` methods return SFrames. These functions can filter edges and vertices based on vertex IDs or attributes. Omitting IDs and attributes returns all vertices or edges.

```
sub_verts = g.get_vertices(ids=['James Bond'])
print sub_verts
```

<code>_id</code>	<code>gender</code>	<code>license_to_kill</code>	<code>villian</code>
James Bond	M	1	0

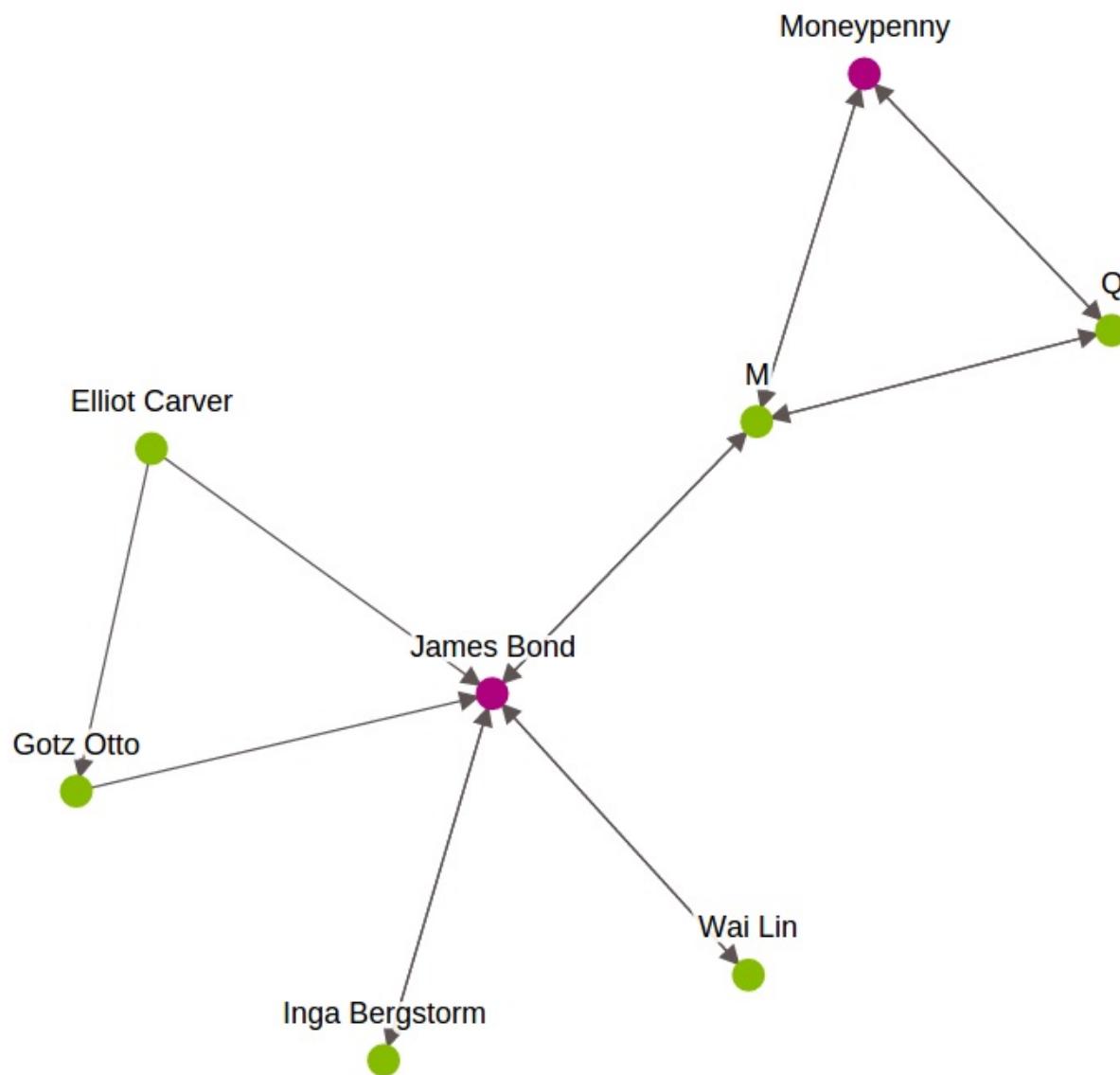
[1 rows x 4 columns]

```
sub_edges = g.get_edges(fields={'relation': 'worksfor'})
print sub_edges
```

```
+-----+-----+-----+
|   __src_id |   __dst_id | relation |
+-----+-----+-----+
|       M    | Moneypenny | worksfor |
|       M    | James Bond  | worksfor |
|       M    |      Q      | worksfor |
| Elliot Carver | Henry Gupta | worksfor |
| Elliot Carver | Gotz Otto  | worksfor |
+-----+-----+-----+
[5 rows x 3 columns]
```

The `get_neighborhood` method provides a convenient way to retrieve the subset of a graph near a set of target vertices, also known as the *egocentric neighborhood* of the target vertices. The `radius` of the neighborhood is the maximum length of a path between any of the targets and a neighborhood vertex. If `full_subgraph` is true, then edges between neighborhood vertices are included even if the edges are not on direct paths between a target and a neighbor.

```
targets = ['James Bond', 'Moneypenny']
subgraph = g.get_neighborhood(ids=targets, radius=1, full_subgraph=True)
subgraph.show(vlabel='id', highlight=['James Bond', 'Moneypenny'], arrows=True)
```



Modifying SGraphs

SGraphs are *structurally immutable*, but the data stored on vertices and edges can be mutated using two special SGraph properties. `sGraph.vertices` and `sGraph.edges` are SFrames containing the vertex and edge data, respectively. The following examples show the difference between the special graph-related SFrames and normal SFrames. First, note that the following lines both produce the same effect.

```

g.edges.print_rows(5)
g.get_edges().print_rows(5)

```

```
+-----+-----+-----+
|   __src_id    |   __dst_id    |   relation   |
+-----+-----+-----+
| Moneypenny   |      M       | managed_by   |
| Inga Bergstorm | James Bond | friend       |
| Moneypenny   |      Q       | colleague    |
| Henry Gupta  | Elliot Carver | killed_by   |
| James Bond   | Inga Bergstorm | friend       |
+-----+-----+-----+
[5 rows x 3 columns]
```

The difference is that the return value of `g.get_edges()` is a normal SFrame independent from `g`, whereas `g.edges` is bound to `g`. We can modify the edge data using this special edge SFrame. The next snippet mutates the relation attribute on the edges of `g`. In particular, it extracts the first letter and converts it to upper case.

```
g.edges['relation'] = g.edges['relation'].apply(lambda x: x[0].upper())
g.get_edges().print_rows(5)
```

```
+-----+-----+-----+
|   __src_id    |   __dst_id    |   relation   |
+-----+-----+-----+
| Moneypenny   |      M       |      M        |
| Inga Bergstorm | James Bond |      F        |
| Moneypenny   |      Q       |      C        |
| Henry Gupta  | Elliot Carver |      K        |
| James Bond   | Inga Bergstorm |      F        |
| ...          | ...          | ...          |
+-----+-----+-----+
[20 rows x 3 columns]
```

On the other hand, the following code does not mutate the relation attribute on the edges of `g`. If it had a permanent effect, the relation field would be converted a lower case letter, but in the result it clearly remains upper case.

```
e = g.get_edges() # e is a normal SFrame independent of g.
e['relation'] = e['relation'].apply(lambda x: x[0].lower())
g.get_edges().print_rows(5)
```

<code>__src_id</code>	<code>__dst_id</code>	<code>relation</code>
Moneypenny	M	M
Inga Bergstrom	James Bond	F
Moneypenny	Q	C
Henry Gupta	Elliot Carver	K
James Bond	Inga Bergstrom	F
...

[20 rows x 3 columns]

Calling a method like `head()`, `tail()`, or `append()` on a special graph-related SFrame also results in a new instance of a regular SFrame. For example, the following code does not mutate `g`.

```
e = g.edges.head(5)
e['is_friend'] = e['relation'].apply(lambda x: x[0] == 'F')
```

Another important difference of these two special SFrames is that the `__id`, `__src_id`, and `__dst_id` fields are not mutable because changing them would change the structure of the graph and SGraph is *structually immutable*.

Otherwise, `g.vertices` and `g.edges` act like normal SFrames, which makes modifying graph data very easy. For example, adding (removing) an edge field is the same as adding (removing) a column to (from) an SFrame:

```
g.edges['weight'] = 1.0
del g.edges['weight']
```

The `triple_apply` method provides a particularly powerful way to modify SGraph vertex and edge attributes. `triple_apply` applies a user-defined function to all edges asynchronously, allowing you to do a computation that modifies edge data based on vertex data, or vice versa. A wide range of methods---[single-source shortest path](#) and [weighted PageRank](#), for example---can be expressed very simply with this primitive.

The first step is to define a function that takes as input an edge in the graph, together with the incident source and destination vertices. This *triple apply function* modifies vertex and edge fields in some way, then returns the modified (source vertex, edge, destination vertex) triple. In this example, we compute the *degree* of each vertex in the James Bond graph, which is the number of edges that touch each vertex.

```
def increment_degree(src, edge, dst):
    src['degree'] += 1
    dst['degree'] += 1
    return (src, edge, dst)
```

The next step is to create a new field in our SGraph's vertex data to hold the answer.

```
g.vertices['degree'] = 0
```

Finally, we use the `triple_apply` method to apply the function to all of the edges (together with their incident source and destination vertices). This method requires specification of which fields are allowed to be changed by the our function.

```
g = g.triple_apply(increment_degree, mutated_fields=['degree'])
print g.vertices.sort('degree', ascending=False)
```

<code>_id</code>	<code>degree</code>	<code>gender</code>	<code>license_to_kill</code>	<code>villian</code>
James Bond	8	M	1	0
Elliot Carver	7	M	0	1
M	6	M	1	0
Moneypenny	4	F	1	0
Q	4	M	1	0
Paris Carver	3	F	0	1
Inga Bergstrom	2	F	0	0
Henry Gupta	2	M	0	1
Wai Lin	2	F	1	0
Gotz Otto	2	M	0	1

[10 rows x 5 columns]

James Bond is quite the popular guy!

To learn more, check out the [graph analytics toolkits](#), the [API Reference](#) for SGraphs, the [hands-on exercises](#) at the end of the chapter.

Time series

For data sources such as usage logs, sensor measurements, and financial instruments, the presence of a time-stamp results in an implicit temporal ordering on the observations.

In these applications, it becomes important to be able to treat the time-stamp as an index around which we can perform several important operations such as:

- **grouping** the data with respect to various intervals of time
- **aggregating** data across time intervals
- **transforming** data into regular discrete intervals
- **windowing** operations based on data from the previous time periods

In this chapter, we will use a dataset obtained from the [UCI machine learning repository](#). The dataset contains measurements of electric power consumption in one household with a one-minute sampling rate over a period of almost 4 years. The entire dataset contains around 2,075,259 measurements gathered between December 2006 and November 2010 (47 months).

Time series construction

The `TimeSeries` object is the fundamental data structure for multivariate time series data. `TimeSeries` objects are backed by a single `SFrame`, but include extra metadata. Each column in the SFrame corresponds to a univariate time series.

T	V_1	V_2	...	V_k
t_1	v_{11}	v_{21}	...	v_{k1}
t_2	v_{12}	v_{22}	...	v_{k2}
t_3	v_{13}	v_{23}	...	v_{k3}
...
...
t_n	v_{1n}	v_{2n}	...	v_{kn}

```
import graphlab as gl
import datetime as dt

household_data = gl.SFrame(
    "https://static.turi.com/datasets/household_electric_sample/household_electric_samp
```

```
Data:
```

Global_active_power	Global_reactive_power	Voltage	DateTime
4.216	0.418	234.84	2006-12-16 17:24:00
5.374	0.498	233.29	2006-12-16 17:26:00
3.666	0.528	235.68	2006-12-16 17:28:00
3.52	0.522	235.02	2006-12-16 17:29:00
3.7	0.52	235.22	2006-12-16 17:31:00
3.668	0.51	233.99	2006-12-16 17:32:00
3.27	0.152	236.73	2006-12-16 17:40:00
3.728	0.0	235.84	2006-12-16 17:43:00
5.894	0.0	232.69	2006-12-16 17:44:00
7.026	0.0	232.21	2006-12-16 17:46:00

[1025260 rows x 4 columns]

We construct a `TimeSeries` object from the SFrame `household_data` by specifying the `DateTime` column as the index column. The data is **sorted** by the `DateTime` column when indexed into a time series.

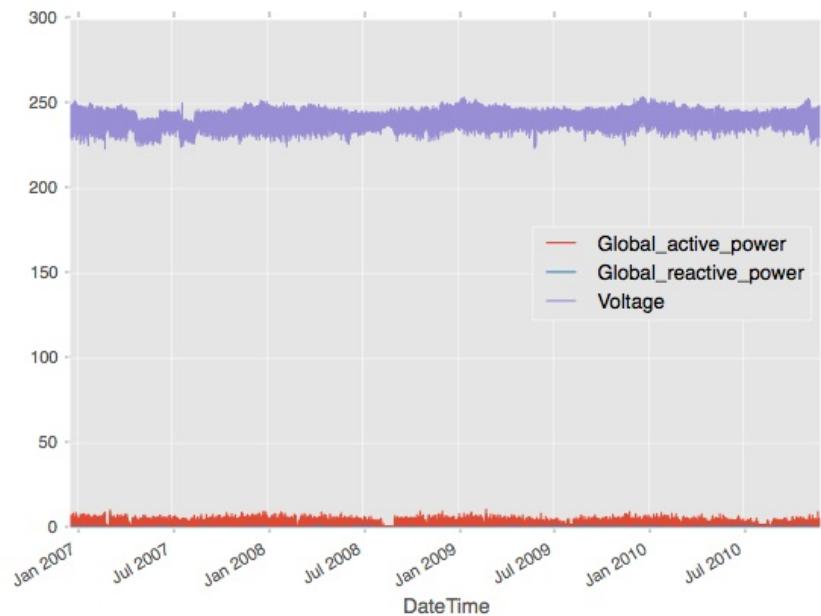
```
household_ts = gl.TimeSeries(household_data, index="DateTime")
```

The index column of the TimeSeries is: `DateTime`

DateTime	Global_active_power	Global_reactive_power	Voltage
2006-12-16 17:24:00	4.216	0.418	234.84
2006-12-16 17:26:00	5.374	0.498	233.29
2006-12-16 17:28:00	3.666	0.528	235.68
2006-12-16 17:29:00	3.52	0.522	235.02
2006-12-16 17:31:00	3.7	0.52	235.22
2006-12-16 17:32:00	3.668	0.51	233.99
2006-12-16 17:40:00	3.27	0.152	236.73
2006-12-16 17:43:00	3.728	0.0	235.84
2006-12-16 17:44:00	5.894	0.0	232.69
2006-12-16 17:46:00	7.026	0.0	232.21

[1025260 rows x 4 columns]

The following figure illustrates the multivariate time series. The index column `DateTime` is the x-axis and the columns `Global_active_power`, `Global_reactive_power`, and `Voltage` are illustrated in the y-axis.



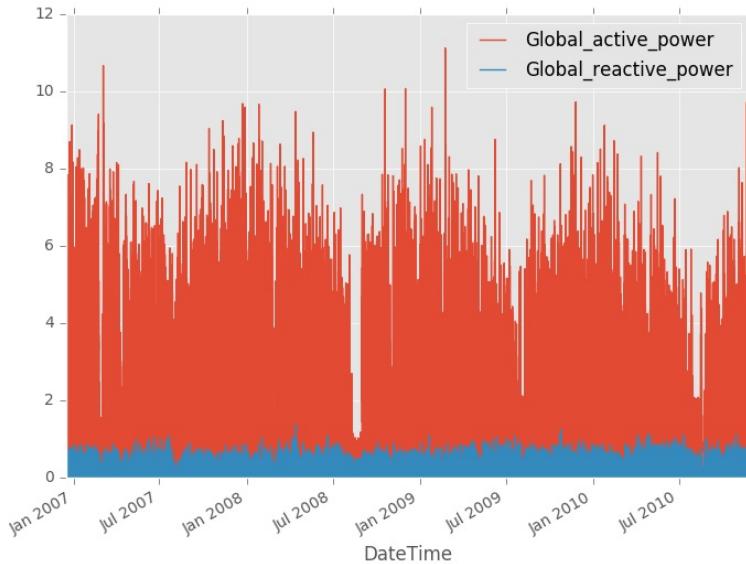
Now, the dataset is indexed by the column `datetime` and all future operations involving time are now optimized. At any point of time, the time series can be **converted to an SFrame** using the `to_sframe` function at **zero cost**.

```
sf = household_ts.to_sframe()
```

Note that each column in the `TimeSeries` object is an **SArray**. A subset of columns can be selected as follows:

```
ts_power = household_ts[['Global_active_power', 'Global_reactive_power']]
```

The following figure illustrates the time series `ts_power`.



Resampling

In many practical time series analysis problems, we require observations to be over uniform time intervals. However, data is often in the form of non-uniform events with accompanying time stamps. As a result, one common prerequisite for time series applications is to convert any time series that is potentially irregularly sampled to one that is sampled at a regular frequency (or to a frequency different from the input data source).

There are three important primitive operations required for this purpose:

- **Mapping** – The operation that determines which time slice a specific observation belongs to.
- **Interpolation/Upsampling** – The operation used to fill in the missing values when there are no observations that map to a particular time slice.
- **Aggregation/Downsampling** –The operation used to aggregate multiple observations that belong to the same time slice.

As an example, we resample the `household_ts` into a time series at an hourly granularity.

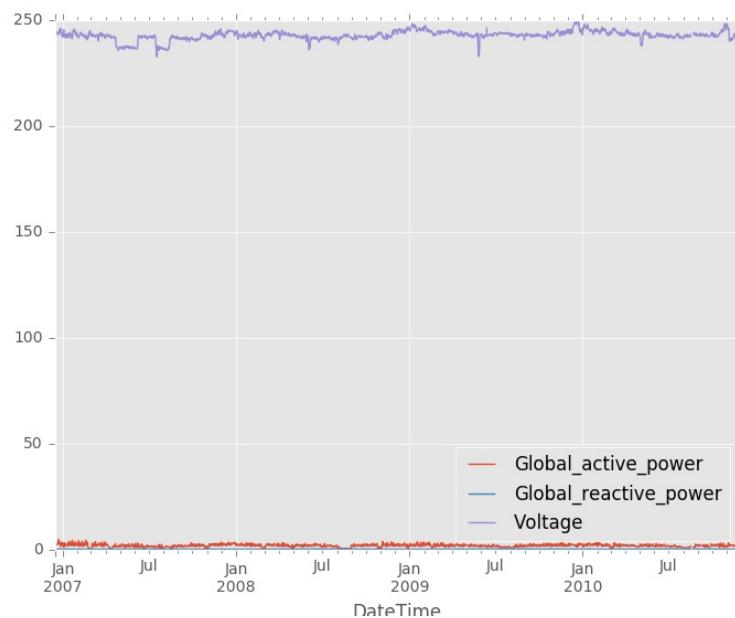
```
import datetime as dt

day = dt.timedelta(days = 1)
daily_ts = household_ts.resample(day, downsample_method='max', upsample_method=None)
```

DateTime	Global_active_power	Global_reactive_power	Voltage
2006-12-16 00:00:00	7.026	0.528	243.73
2006-12-17 00:00:00	6.58	0.582	249.07
2006-12-18 00:00:00	5.436	0.646	248.48
2006-12-19 00:00:00	7.84	0.606	248.89
2006-12-20 00:00:00	5.988	0.482	249.48
2006-12-21 00:00:00	5.614	0.688	247.08
2006-12-22 00:00:00	7.884	0.622	248.82
2006-12-23 00:00:00	8.698	0.724	246.77
2006-12-24 00:00:00	6.498	0.494	249.27
2006-12-25 00:00:00	6.702	0.7	250.62

[1442 rows x 4 columns]

The following figure illustrates the resampled time series `daily_ts`.



In this example, the **mapping** is performed by choosing intervals of length **1 hour**, the **downsampling** method is chosen by returning the **maximum** value (for each column) of all the data points in the original time series, the **upsampling** method sets a `None` value (for a column) corresponding to an interval in the returned time series if there are no any values (for that column) within that time interval in the original time series.

Shifting time series data

Time series data can also be shifted along the time dimension using the `TimeSeries.shift` and `TimeSeries.tshift` methods.

The `tshift` operator shifts the index column of the time series along the time dimension while keeping other columns intact. For example, we can shift the `household_ts` by 5 minutes, so all the tuples by an hour:

```
interval = dt.timedelta(hours = 1)
shifted_ts = household_ts.tshift(interval)
```

DateTime	Global_active_power	Global_reactive_power	Voltage
2006-12-16 18:24:00	4.216	0.418	234.84
2006-12-16 18:26:00	5.374	0.498	233.29
2006-12-16 18:28:00	3.666	0.528	235.68
2006-12-16 18:29:00	3.52	0.522	235.02
2006-12-16 18:31:00	3.7	0.52	235.22
2006-12-16 18:32:00	3.668	0.51	233.99
2006-12-16 18:40:00	3.27	0.152	236.73
2006-12-16 18:43:00	3.728	0.0	235.84
2006-12-16 18:44:00	5.894	0.0	232.69
2006-12-16 18:46:00	7.026	0.0	232.21

[1025260 rows x 8 columns]

The `shift` operator shifts forward/backward all the value columns while keeping the index column intact. Notice that this operator does not change the *range* of the TimeSeries object and it fills those edge tuples that lost their value with `None`.

```
shifted_ts = household_ts.shift(steps = 3)
```

DateTime	Global_active_power	Global_reactive_power	Voltage
2006-12-16 17:24:00	None	None	None
2006-12-16 17:26:00	None	None	None
2006-12-16 17:28:00	None	None	None
2006-12-16 17:29:00	4.216	0.418	234.84
2006-12-16 17:31:00	5.374	0.498	233.29
2006-12-16 17:32:00	3.666	0.528	235.68
2006-12-16 17:40:00	3.52	0.522	235.02
2006-12-16 17:43:00	3.7	0.52	235.22
2006-12-16 17:44:00	3.668	0.51	233.99
2006-12-16 17:46:00	3.27	0.152	236.73

[1025260 rows x 8 columns]

Index Join

Another important feature of TimeSeries objects in GraphLab Create is the ability to efficiently join them across the index column. So far we created a resampled TimeSeries from one of the electric meters. Now is the time to join the first resampled TimeSeries object with the second TimeSeries object.

```
sf_other = gl.SFrame(
    'https://static.turi.com/datasets/household_electric_sample/household_electric_samp
ts_other = gl.TimeSeries(sf_other, index = 'DateTime')
household_ts.index_join(ts_other, how='inner')
```

DateTime	Global_active_power	Global_reactive_power	Voltage
2006-12-16 17:24:00	4.216	0.418	234.84
2006-12-16 17:26:00	5.374	0.498	233.29
2006-12-16 17:28:00	3.666	0.528	235.68
2006-12-16 17:29:00	3.52	0.522	235.02
2006-12-16 17:31:00	3.7	0.52	235.22
2006-12-16 17:32:00	3.668	0.51	233.99
2006-12-16 17:40:00	3.27	0.152	236.73
2006-12-16 17:43:00	3.728	0.0	235.84
2006-12-16 17:44:00	5.894	0.0	232.69
2006-12-16 17:46:00	7.026	0.0	232.21
<hr/>			
Global_intensity			
18.4			
23.0			
15.8			
15.0			
15.8			
15.8			
13.8			
16.4			
25.4			
30.6			
<hr/>			
[1025260 rows x 5 columns]			

The `how` parameter in `index_join` operator determines the join method. The acceptable values are 'inner', 'left', 'right', and 'outer'. The behavior is exactly like the **SFrame** join methods.

Time series slicing

The range of a time series is defined as the interval `(start, end)` of the time stamps that span the time series. It can be obtained as follows:

```
start_time, end_time = household_ts.range
```

```
(datetime.datetime(2006, 12, 16, 17, 24), datetime.datetime(2007, 11, 26, 20, 57))
```

We can obtain a slice of a time series that lies within its range using the `TimeSeries.slice` operator.

```
import datetime as dt
start = dt.datetime(2006, 12, 16, 17, 24)
end = dt.datetime(2007, 11, 26, 21, 2)

sliced_ts = household_ts.slice(start, end)
```

DateTime	Global_active_power	Global_reactive_power	Voltage
2006-12-16 17:24:00	4.216	0.418	234.84
2006-12-16 17:26:00	5.374	0.498	233.29
2006-12-16 17:28:00	3.666	0.528	235.68
2006-12-16 17:29:00	3.52	0.522	235.02
2006-12-16 17:31:00	3.7	0.52	235.22
2006-12-16 17:32:00	3.668	0.51	233.99
2006-12-16 17:40:00	3.27	0.152	236.73
2006-12-16 17:43:00	3.728	0.0	235.84
2006-12-16 17:44:00	5.894	0.0	232.69
2006-12-16 17:46:00	7.026	0.0	232.21

[246363 rows x 4 columns]

We can also `slice` the data for a particular year as follows:

```
start = dt.datetime(2010, 1, 1)
end = dt.datetime(2011, 1, 1)
ts_2010 = household_ts.slice(start, end)
```

DateTime	Global_active_power	Global_reactive_power	Voltage
2010-01-01 00:00:00	1.79	0.236	240.65
2010-01-01 00:01:00	1.78	0.234	240.07
2010-01-01 00:03:00	1.746	0.186	240.26
2010-01-01 00:06:00	1.68	0.1	239.72
2010-01-01 00:07:00	1.688	0.102	240.34
2010-01-01 00:08:00	1.676	0.072	241.0
2010-01-01 00:11:00	1.618	0.0	240.11
2010-01-01 00:13:00	1.618	0.0	240.09
2010-01-01 00:14:00	1.622	0.0	240.38
2010-01-01 00:15:00	1.622	0.0	240.4

[229027 rows x 4 columns]

Time series grouping

Quite often in time series analysis, we are required to split a single large time series into groups of smaller time series grouped based on a property of the time stamp (e.g. per day of week).

The output of this operator is a `graphlab.timeseries.GroupedTimeSeries` object, which can be used for retrieving one or more groups, or iterating through all groups. Each group is a separate time series which possesses the same columns as the original time series.

In this example, we group the time series `household_ts` by the day of the week.

```
household_ts_groups = household_ts.group(gl.TimeSeries.date_part.WEEKDAY)
print household_ts_groups.groups()
```

```
Rows: 7
[0, 1, 2, 3, 4, 5, 6]
```

`household_ts_groups` is a `GroupedTimeSeries` containing 7 groups where each group is a single `TimeSeries`. In this example groups are named between 0 and 6 where 0 is Monday. We can access the data corresponding to a Monday as follows:

```
household_ts_monday = household_ts_groups.get_group(0)
```

DateTime	Global_active_power	Global_reactive_power	Voltage
2006-12-18 00:00:00	0.278	0.126	246.17
2006-12-18 00:03:00	0.206	0.0	245.94
2006-12-18 00:04:00	0.206	0.0	245.98
2006-12-18 00:06:00	0.204	0.0	245.22
2006-12-18 00:07:00	0.204	0.0	244.14
2006-12-18 00:08:00	0.212	0.0	244.0
2006-12-18 00:09:00	0.316	0.134	244.62
2006-12-18 00:10:00	0.308	0.132	244.61
2006-12-18 00:11:00	0.306	0.134	244.97
2006-12-18 00:12:00	0.306	0.136	245.51

[146934 rows x 4 columns]

We can also iterate over all the groups in this GroupedTimeSeries object:

```
for name, group in household_ts_groups:
    print name, group
```

Time series union

We can also merge multiple time series into a single one using the `union` operator. The merged time series is a valid time series with the time stamps sorted correctly. In this example, we will use the `union` operator to re-unite the time series that we split by the day of the week (using the `group` operator).

```
household_ts_combined = household_ts_groups.get_group(0)
for i in range(1, 7):
    group = household_ts_groups.get_group(i)
    household_ts_combined = household_ts_combined.union(group)
```

DateTime	Global_active_power	Global_reactive_power	Voltage
2006-12-16 17:24:00	4.216	0.418	234.84
2006-12-16 17:26:00	5.374	0.498	233.29
2006-12-16 17:28:00	3.666	0.528	235.68
2006-12-16 17:29:00	3.52	0.522	235.02
2006-12-16 17:31:00	3.7	0.52	235.22
2006-12-16 17:32:00	3.668	0.51	233.99
2006-12-16 17:40:00	3.27	0.152	236.73
2006-12-16 17:43:00	3.728	0.0	235.84
2006-12-16 17:44:00	5.894	0.0	232.69
2006-12-16 17:46:00	7.026	0.0	232.21

[1025260 rows x 4 columns]

Rolling Statistics

Rolling aggregates, also known as a **moving window aggregates** or **running aggregates**, aggregate statistics from observations in a window of observations that are before/after the current point. With this feature, you can compute:

- For each observation, an aggregate value (mean, variance, count etc.) of a fixed number of observations that occur before the current observation.
- A similar aggregation operation on observations that occur after the current observation.
- A combination of the above two operations.

The subset of the observations on which the aggregation is performed is defined as an inclusive range relative to the position to each observation. The `window_start` and `window_end` are the two parameters that define the start of the window (relative to the current observation) and the end of the window (relative to the current observation) over which the aggregations are performed.

For example, we can compute aggregates on:

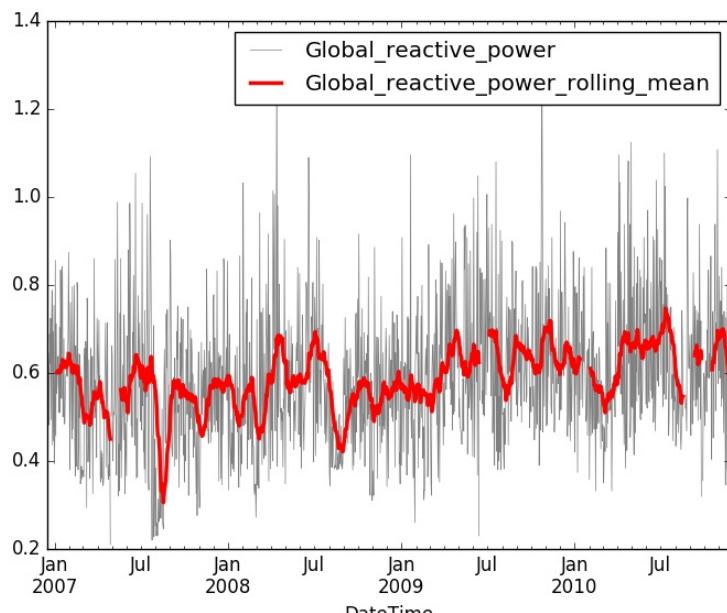
- the previous 5 observations including the current using `window_start = -5` and `window_end = 0`.
- the next 5 observations including the current using `window_start = 0` and `window_end = 5`.
- the previous 5 observations excluding the current using `window_start = -5` and `window_end = -1`.
- the next 5 observations excluding the current using `window_start = 1` and `window_end = 5`.

```
daily_ts['Global_reactive_power_rolling_mean'] = \
    daily_ts['Global_reactive_power'].rolling_mean(-20, 0)
daily_ts[['Global_reactive_power_rolling_mean', 'Global_reactive_power']].print_rows(50)
```

DateTime	Global_reactive_power_rolling_mean	Global_reactive_power
2006-12-16 00:00:00	None	0.528
2006-12-17 00:00:00	None	0.582
2006-12-18 00:00:00	None	0.646
2006-12-19 00:00:00	None	0.606
2006-12-20 00:00:00	None	0.482
2006-12-21 00:00:00	None	0.688
2006-12-22 00:00:00	None	0.622
2006-12-23 00:00:00	None	0.724
2006-12-24 00:00:00	None	0.494
2006-12-25 00:00:00	None	0.7
2006-12-26 00:00:00	None	0.788
2006-12-27 00:00:00	None	0.436
2006-12-28 00:00:00	None	0.694
2006-12-29 00:00:00	None	0.446
2006-12-30 00:00:00	None	0.754
2006-12-31 00:00:00	None	0.394
2007-01-01 00:00:00	None	0.454
2007-01-02 00:00:00	None	0.856
2007-01-03 00:00:00	None	0.458
2007-01-04 00:00:00	None	0.712
2007-01-05 00:00:00	0.600857142857	0.554
2007-01-06 00:00:00	0.603047619048	0.574
2007-01-07 00:00:00	0.604952380952	0.622
2007-01-08 00:00:00	0.604571428571	0.638
2007-01-09 00:00:00	0.604095238095	0.596
2007-01-10 00:00:00	0.605428571429	0.51
2007-01-11 00:00:00	0.60619047619	0.704
2007-01-12 00:00:00	0.600857142857	0.51
2007-01-13 00:00:00	0.606095238095	0.834
2007-01-14 00:00:00	0.616666666667	0.716
2007-01-15 00:00:00	0.609142857143	0.542
2007-01-16 00:00:00	0.611714285714	0.842
2007-01-17 00:00:00	0.630095238095	0.822
2007-01-18 00:00:00	0.620857142857	0.5
2007-01-19 00:00:00	0.624761904762	0.528
2007-01-20 00:00:00	0.621142857143	0.678
2007-01-21 00:00:00	0.625428571429	0.484
2007-01-22 00:00:00	0.626952380952	0.486
2007-01-23 00:00:00	0.612571428571	0.554
2007-01-24 00:00:00	0.619142857143	0.596
2007-01-25 00:00:00	0.613619047619	0.596
2007-01-26 00:00:00	0.618095238095	0.648
2007-01-27 00:00:00	0.618952380952	0.592

2007-01-28 00:00:00	0.6186666666667	0.616	
2007-01-29 00:00:00	0.621238095238	0.692	
2007-01-30 00:00:00	0.634476190476	0.874	
2007-01-31 00:00:00	0.644380952381	0.718	
2007-02-01 00:00:00	0.634380952381	0.494	
2007-02-02 00:00:00	0.62819047619	0.38	
2007-02-03 00:00:00	0.6246666666667	0.76	

The result of the rolling mean is typically smoother than the original curve while still capturing some of the recent trends in the data.



In addition to mean, you can perform the following operations

- Rolling sum
- Rolling variance
- Rolling standard deviation
- Rolling counts
- Rolling min
- Rolling max

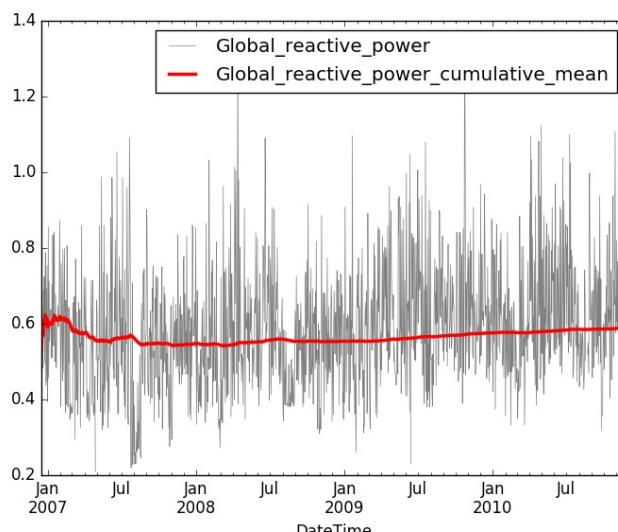
Cumulative Statistics

In addition to rolling aggregates, we can also perform cumulative aggregates using all the observations before (and including) the current observation.

```
daily_ts['Global_reactive_power_cumulative_mean'] = daily_ts['Global_reactive_power'].cumsum()
daily_ts[['Global_reactive_power_cumulative_mean', 'Global_reactive_power']]
```

DateTime	Global_reactive_power_cumu...	Global_reactive_power
2006-12-16 00:00:00	0.528	0.528
2006-12-17 00:00:00	0.555	0.582
2006-12-18 00:00:00	0.585333333333	0.646
2006-12-19 00:00:00	0.5905	0.606
2006-12-20 00:00:00	0.5688	0.482
2006-12-21 00:00:00	0.588666666667	0.688
2006-12-22 00:00:00	0.593428571429	0.622
2006-12-23 00:00:00	0.60975	0.724
2006-12-24 00:00:00	0.596888888889	0.494
2006-12-25 00:00:00	0.6072	0.7

The result of the cumulative mean is typically much smoother than the original curve. Unlike the rolling means, the cumulative mean cannot capture recent trends in the data, but it can however spot global trends in the data.



In addition to mean, you can perform the following operations

- Cumulative sum
- Cumulative variance
- Cumulative standard deviation
- Cumulative counts
- Cumulative min
- Cumulative max

Operations common with SFrame/SArray

Because the time series data structure is backed by an SFrame, there are many operations that behave exactly like the SFrame. These include

- Logical filters (row selection)
- SArray apply functions (univariate user defined functions UDFs)
- Time series apply functions (multivariate UDFs)
- Selecting columns
- Adding, removing, and swapping columns
- Head, tail, row range selection
- Joins (on the non-index column)

See the chapter on SFrame for more usage details on the above functions.

Save and Load

Just like every other object, the time series can be saved and loaded as follows:

```
household_ts.save("/tmp/first_copy")
household_ts_copy = graphlab.TimeSeries("/tmp/first_copy")
```

Visualizing Data

Data visualization can help you explore, understand, and gain insight from your data. Visualization can complement other methods of data analysis by taking advantage of the human ability to recognize patterns in visual information.

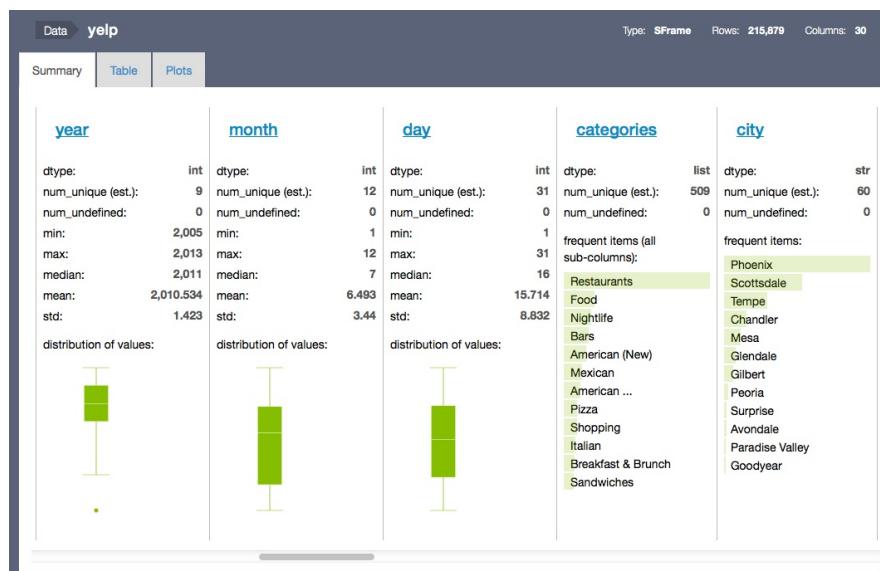
GraphLab Canvas provides an interactive browser-based visualization platform to explore your data. The data structures that support visualization in Canvas include `SFrame`, `SArray`, and `SGraph`. Each of these data structures can be shown in Canvas by calling the `show` method on an instance of one of those types. Canvas supports two render targets: `'browser'` (the interactive browser-based experience) and `'ipynb'` (visualizations are embedded in an IPython Notebook or Jupyter Notebook output cell).

SFrame Visualization

Data in an SFrame can be visualized with `SFrame.show()`. Table and Summary are the two types of visualizations¹ for SFrame, each represented by a tab in the Canvas user interface. The Table view provides a scrollable, interactive tabular view of the data inside the SFrame. Like the SFrame itself, the Table view can scale to as much data as will fit on a disk -- only the rows being viewed are loaded. The paging control on the left side of the view allows you to move quickly through the SFrame or jump to a particular row.

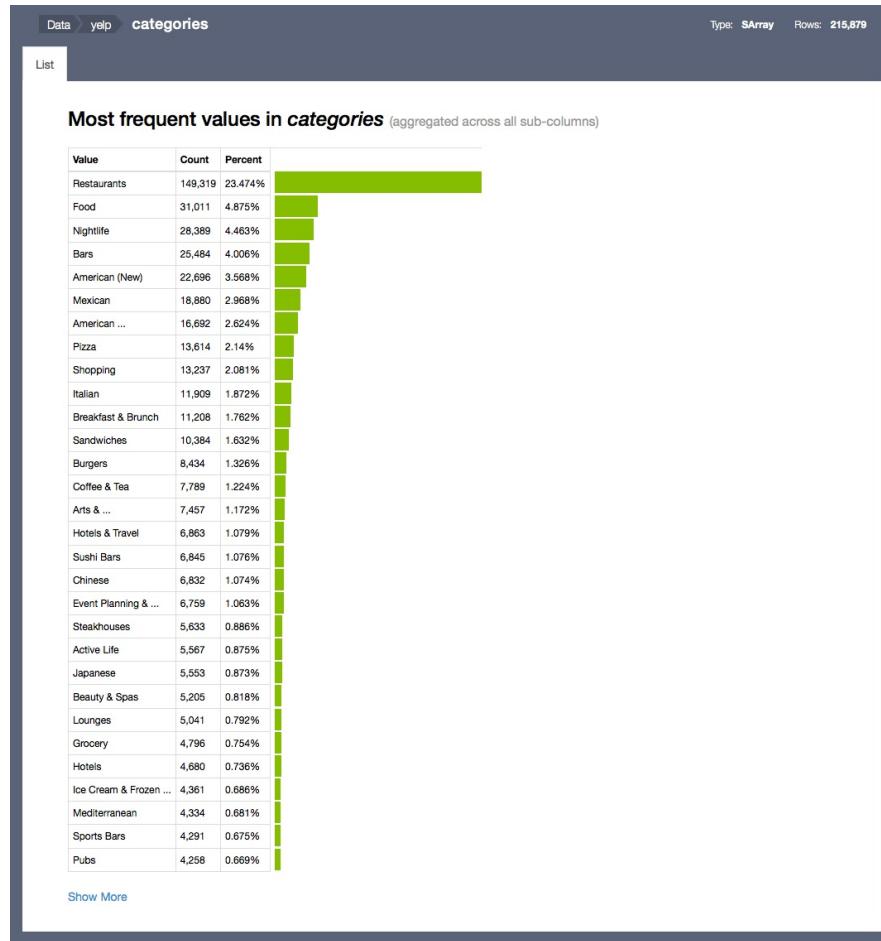
Data yelp						Type: SFrame	Rows: 215,879	Columns: 30
	Summary	Table	Plots	year	month	day	categories	city
row #	0			2,011	3	4	["Hot Dogs", "Food ...	Phoenix
				2,011	2	5	["Greek", ...	Scottsdale
				2,012	4	20	["Breakfast & ...	Gila Bend
				2,012	3	16	["Active Life", ...	Tempe
				2,011	6	14	["Bakeries", ...	Chandler
				2,010	3	15	["Thai", ...	Tempe
				2,006	11	29	["Mexican", ...	Phoenix
				2,012	2	21	["Breakfast & ...	Phoenix
				2,010	3	27	["Thai", ...	Mesa
				2,011	6	9	["Italian", ...	Scottsdale
				2,008	9	4	["Seafood", ...	Scottsdale
				2,012	2	21	["Food", "Tea ...	Glendale
				2,011	9	14	["Hotels & ...	Phoenix
				2,011	7	24	["Breakfast & ...	Scottsdale
				2,011	12	29	["Thai", ...	Chandler
				2,012	5	9	["Thai", ...	Chandler
				2,010	8	16	["Food", "Health ...	Tempe
				2,012	2	28	["Vegetarian", ...	Tempe
				2,010	1	16	["Vietnamese", ...	Phoenix
				2,008	8	19	["Latin American", ...	Phoenix
				2,011	7	16	["Men's Clothing", ...	Tempe
				2,007	12	11	["Vegetarian", ...	Phoenix
				2,012	8	6	["Italian", ...	Phoenix
				215,878				

The Summary view shows which columns are in the SFrame, with a summary of the data inside each column. Numeric column types (`int` and `float`) show a [box plot](#), `str` columns show a table of the most frequently occurring items in the column, and recursive column types (`dict`, `list`, and `array`) show some combination of those plots depending on the type of the underlying data.

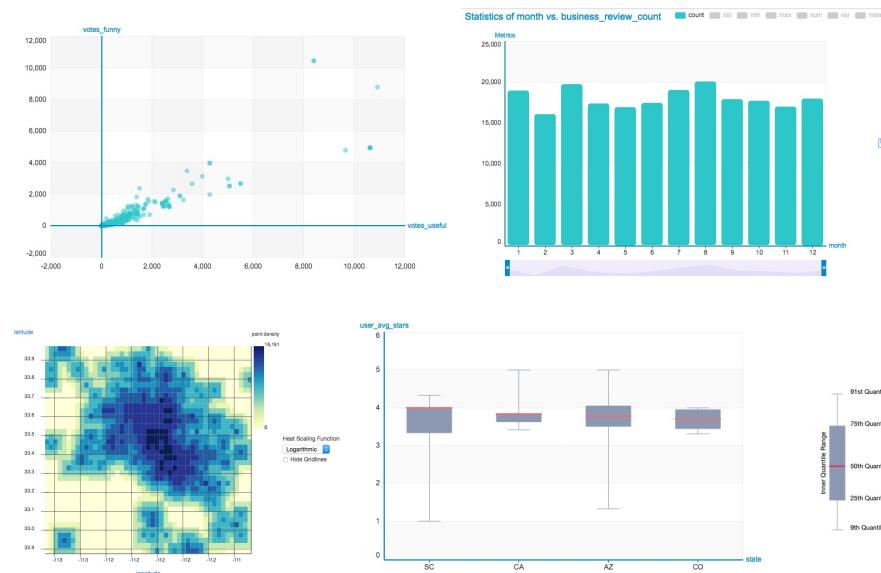


The summary view supports drill-down visualization that any column within it can be visualized by clicking on the corresponding title. For example, clicking on "categories" brings up an SArray view. Below we have more information about how to visualize an SArray

programmatically.



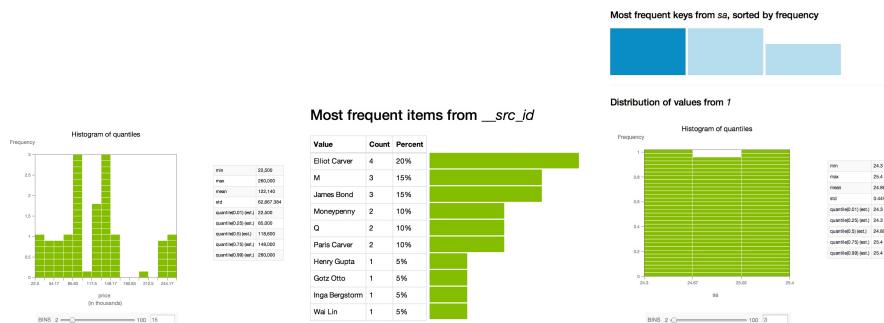
SFrame can also be visualized using integrated bi-variate plot types, currently we support Scatter plot, Heatmap, Bar chart, Box plot, and etc. The plot types can either be specified over [API](#) or explored under the "Plots" tab.



SArray Visualization

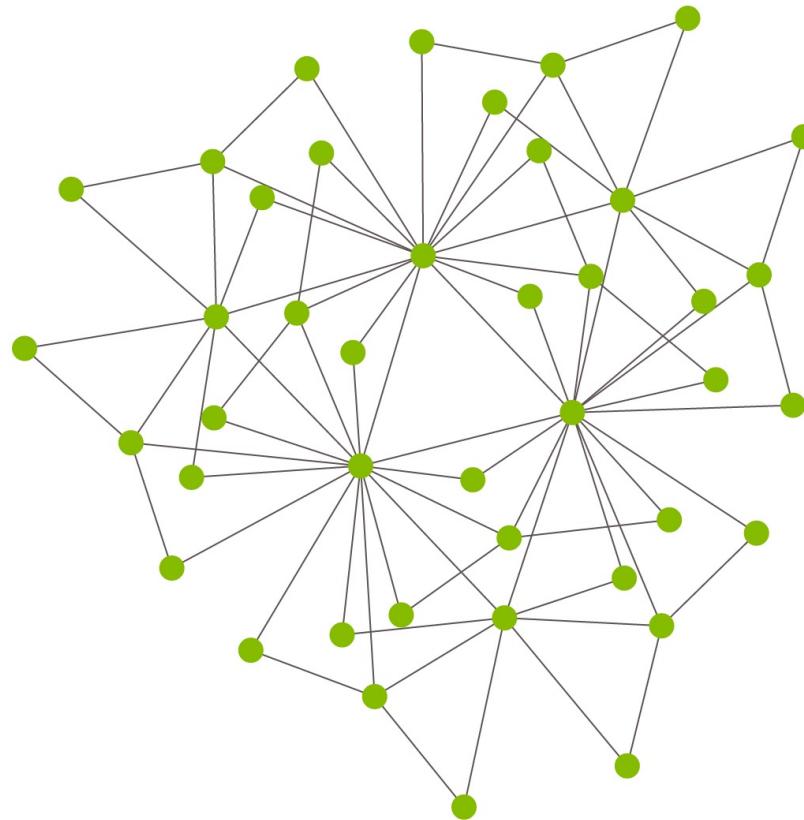
Data in an SArray can be visualized with `SArray.show()`. Canvas has a specialized visualization for each supported `dtype` in `SArray`. The visualization types currently supported (based on column `dtype`) are:

<code>dtype</code>	Visualization
<code>float</code>	Histogram of quantiles (approximated histogram based on <code>sketch_summary</code> quantiles)
<code>int</code>	Histogram of quantiles and (if there are any items with >0.01% occurrence) table of most frequent items
<code>str</code>	Table of most frequent items
<code>array</code>	Histogram of quantiles (aggregate or per-subcolumn)
<code>list</code>	Table of most frequent items (aggregate)
<code>dict</code>	Filterable table of most frequent keys, with aggregate or filtered values visualized according to value <code>dtype</code>



SGraph Visualization

Graph structure can be visualized with `SGraph.show()`. The vertices and edges are laid out in a two-dimensional plot, with vertices drawn as circles and edges as lines between them. The default layout algorithm is force-directed, but it is possible to apply a custom layout algorithm with the `vertex_positions` parameter. In general, `SGraph.show` offers parameterization of the size, position, and color of vertices and edges, which can be used to great effect on graphs representing different types of data.



Model Visualization

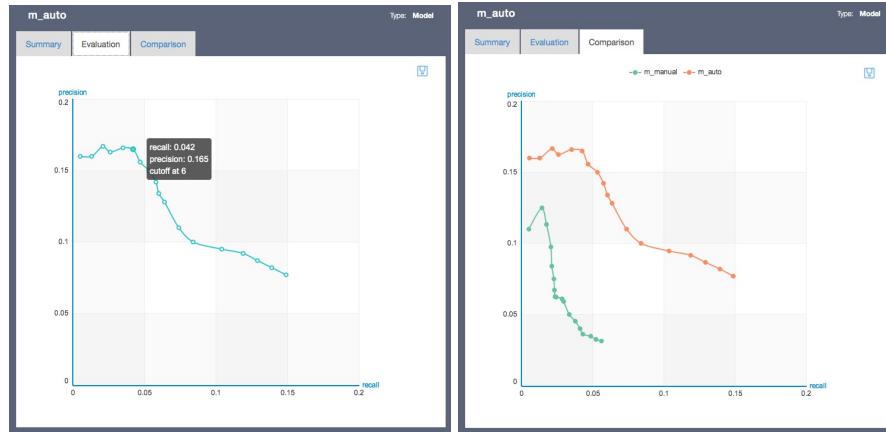
Models such as Recommender and Classifier can be visually inspected by calling `model.show()`. The model visualization provides model summary view, model evaluation view and model comparison view. More details about how to use the model comparison visualization can be found in `gl.compare()` and `gl.show_comparison()`.

The summary view provides basic statistics about the training data as well as the model

m_auto		Type: Model	
	Summary	Evaluation	Comparison
Schema			
User ID			
User ID			user_id
Item ID			item_id
Target			rating
Additional observation features			1
Number of user side features			0
Number of item side features			0
Statistics			
Number of observations			9997457
Number of users			69878
Number of items			10677
Training summary			
Training time			526.8901

training time.

The model evaluation provide model specific evaluation metrics such as precision-recall. The hover tooltip shows more details about the model performance at a specific cutoff value. The model comparison view shows multiple models in the same view space and offers interactive highlighting to support focused analysis.



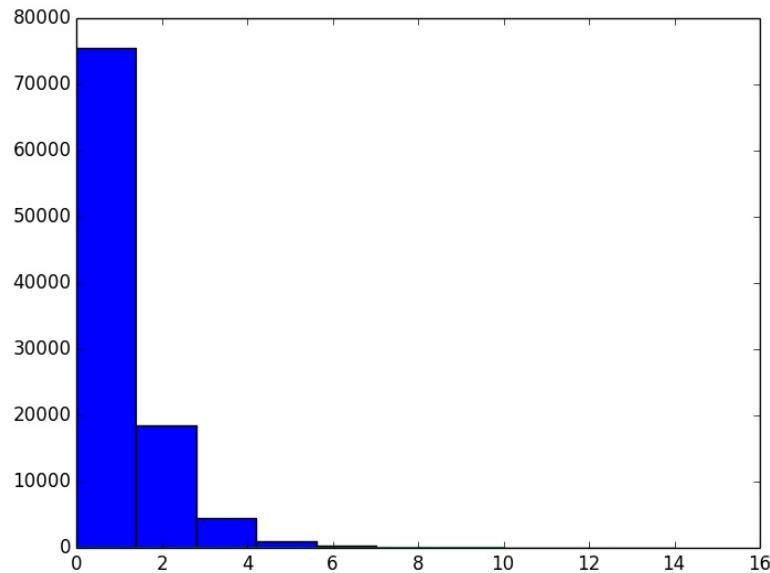
Integration with other visualization tools

There are many other software tools that can help with different types of data visualization. Integrating with GraphLab Create is easy, since we provide methods on `SFrame`, `SArray`, and `SGraph` to transform and retrieve the underlying data as native Python types. One commonly used Python module for visualization is `matplotlib`.

Here is an example of a histogram displayed with `matplotlib`. For this particular use, it might be easier to simply call `show` on a numeric `SArray`, which will also give a histogram -- this example is intended to demonstrate interoperability with other Python visualization packages.

```
import numpy as np
from matplotlib import pyplot as plt

sa = gl.SArray([np.random.gamma(1) for _ in range(100000)])
n, bins, patches = plt.hist(list(sa))
plt.plot(bins)
plt.show()
```



¹. Note: in the `'ipython'` target, only the Summary visualization of SFrame is supported.



Introduction to Feature Engineering

Feature engineering is an important part of designing an effective machine learning pipeline. It is best described as the process of transforming data from its raw form to something more useful to the predictive model. This can result in much better results on your task.

The basic idea is to construct an object, fit it to a dataset, and transform any new data.

```
# Construct a transformer
sf = gl.SFrame({'docs': ["This is a document!", "This one's also a document."]})
f = graphlab.feature_engineering.TFIDF(features = ['docs'])

# Fit it to a dataset
f.fit(sf)

# Now the object is ready to transform new data
f.transform(sf)
```

Feature engineering objects can be [combined into pipelines](#) and deployed on predictive services (see below for more). There is also a helper function `fit_transform` method that combines the last two methods.

GLC has a collection of feature engineering objects are helpful for transforming SFrames of various types. These feature engineering tasks are grouped based on the feature types:

Numeric Features

- [Quadratic Features](#)
- [Feature Binning](#)
- [Numeric Imputer](#)

Categorical Features

- [One Hot Encoder](#)
- [Count Thresholder](#)
- [Categorical Imputer](#)
- [Count Featurizer](#)

Image Features

- [Deep Feature Extractor](#)

Text features

- [TF-IDF](#)
- [Tokenizer](#)
- [RareWordTrimmer](#)
- [BM25](#)

Misc.

- [Hasher](#)
- [Random Projection](#)
- [Transformer Chain](#)
- [Custom Transformer](#)

Transforming single columns

Many of the above transformations have a corresponding one-liner function whose input is an SArray and output is an SArray. Internally it simply runs `fit_transform` on the corresponding transformation.

```
tfidf_transforms = gl.text_analytics.tf_idf(data['docs'])
bag_of_words_transforms = gl.text_analytics.count_words(data['docs'])
bag_of_ngrams_transforms = gl.text_analytics.count_words(data['docs'])
```

Transforming multiple columns

TF-IDF is an example of a feature engineering object that performs a transformation for each feature (i.e. column name) provided in the `features` argument. Other one-to-one transformations include [CategoricalImputer](#), [CountThreshold](#), [FeatureBinner](#), [NGramCounter](#), [NumericImputer](#), [Tokenizer](#), [WordCounter](#). If you would prefer to have each transformed column be *included* in the SFrame (rather than replacing the original column) you can use the `column_name_prefix` argument to add a prefix the set of transformed columns.

Other transformations take a set of columns and create a single column. Examples include [FeatureHasher](#), [OneHotEncoder](#), and [QuadraticFeatures](#). You may change the name of the output column using the `output_column_name` argument.

Finally, the [RandomProjection](#) tool does a "many-to-many" transformation. It takes a set of columns as input and returns a smaller set of (randomly projected) columns.

Deploying feature engineering transformations

The feature engineering toolkit also makes it easy to deploy your feature engineering models and pipelines.

Suppose we have a simple tokenizer:

```
import graphlab as gl
data = gl.SFrame({'docs': ["This is a document", "Another doc"]})
m = gl.feature_engineering.Tokenizer(features=['docs'])
m.fit(data)
```

Now suppose we have created a Predictive Service object `ps`.

(For more on that, see the [Predictive Services](#) chapter of the user guide.) Then we can take a feature engineering model and add it as a service, apply those changes, and query the model that has been deployed as a service.

```
ps = gl.deploy.predictive_service.load(my_ps_url)
ps.add('my_transformation', m)
ps.apply_changes()

d = [row for row in data] # Create JSON-serializable version of data
ps.query('my_transformation', method='transform', data={'data': d})
```

The resulting JSON will have a "response" field containing the data transformed by the deployed tokenizer model.

```
{u'from_cache': False,
 u'model': u'chris_tmp_wordcounter',
 u'response': [{u'docs': [u'This', u'is', u'a', u'document']},
   {u'docs': [u'Another', u'doc']}],
 u'uuid': u'6bb3627b-708d-4398-9afdb13dd170d8e3',
 u'version': 0}
```

Feedback

Feedback about the feature engineering toolkit is very welcome. Please post questions and comments on our forum or send a note to support@turi.com.

Numeric Features

These transformations are useful when you have numeric data.

- [Quadratic Features](#)
- [Feature Binning](#)
- [Numeric Imputer](#)

Quadratic Features

Adding interaction terms is a good way of injecting complex relationships between predictor variables while still using a simple learning algorithm (ie. Logistic Regression) that is easy to use and explain. The QuadraticFeatures transformer accomplishes this by taking a row of the SFrame, and multiplying the specified features together. If the features are of array.array or dictionary type, multiplications of all possible numeric pairs are computed. Supported types are int, float, array.array, and dict.

When the transformer is applied, an additional column with name specified by ‘output_column_name’ is added to the input SFrame. In this column of dictionary type, interactions are specified in the key names (by concatenating column names and keys/indices if applicable) and values are the multiplied values.

Introductory Example

```
from graphlab.toolkits.feature_engineering import *

# Construct a quadratic features transformer with default options.
sf = graphlab.SFrame({'a': [1,2,3], 'b' : [2,3,4], 'c': [9,10,11]})
quadratic = graphlab.feature_engineering.create(sf, QuadraticFeatures())

# Transform the data.
quadratic_sf = quadratic.transform(sf)

# Save the transformer.
quadratic.save('save-path')

# Compute interactions only for a single column 'a'.
quadratic = graphlab.feature_engineering.create(sf,
                                                QuadraticFeatures(features = ['a']))

# Compute interactions for all columns except 'a'.
quadratic = graphlab.feature_engineering.create(sf,
                                                QuadraticFeatures(excluded_features = ['a']))
```

Fitting and transforming

Once a QuadraticFeatures object is constructed, it must first be fitted, and then the transform function can be called to generate hashed features.

For numeric columns:

```

sf = graphlab.SFrame({'a' : [1,2,3], 'b' : [2,3,4]})
quadratic = graphlab.feature_engineering.QuadraticFeatures()
fit_quadratic = quadratic.fit(sf)
quadratic_sf = fit_quadratic.transform(sf)

```

Columns:

a	int
b	int
quadratic_features	dict

Rows: 3

Data:

a	b	quadratic_features
1	2	{'a': 2, 'a, a': 1, 'b,...': 1}
2	3	{'a': 6, 'a, a': 4, 'b,...': 2}
3	4	{'a': 12, 'a, a': 9, 'b,...': 3}

[3 rows x 3 columns]

For vector columns:

```

l1 = [1,2,3]
l2 = [2,3,4]
sf = graphlab.SFrame({'a' : [l1,l1,l1], 'b' : [l2,l2,l2]})
quadratic = graphlab.feature_engineering.QuadraticFeatures()
fit_quadratic = quadratic.fit(sf)
quadratic_sf = fit_quadratic.transform(sf)

```

Columns:

a	array
b	array
quadratic_features	dict

Rows: 3

Data:

a	b	quadratic_features
[1.0, 2.0, 3.0]	[2.0, 3.0, 4.0]	{'b:0, b:0': 4.0, 'b:0, b:...': 1}
[1.0, 2.0, 3.0]	[2.0, 3.0, 4.0]	{'b:0, b:0': 4.0, 'b:0, b:...': 2}
[1.0, 2.0, 3.0]	[2.0, 3.0, 4.0]	{'b:0, b:0': 4.0, 'b:0, b:...': 3}

[3 rows x 3 columns]

For dictionary columns:

```
dict1 = {'a' : 1 , 'b' : 2 , 'c' : 3}
dict2 = {'d' : 4 , 'e' : 5 , 'f' : 6}
sf = graphlab.SFrame({'a' : [dict1, dict1, dict1], 'b' : [dict2, dict2, dict2]})  
quadratic = graphlab.feature_engineering.QuadraticFeatures()  
fit_quadratic = quadratic.fit(sf)  
quadratic_sf = fit_quadratic.transform(sf)
```

Columns:

a	dict
b	dict
quadratic_features	dict

Rows: 3

Data:

	a	b
1	{'a': 1, 'c': 3, 'b': 2}	{'e': 5, 'd': 4, 'f': 6}
2	{'a': 1, 'c': 3, 'b': 2}	{'e': 5, 'd': 4, 'f': 6}
3	{'a': 1, 'c': 3, 'b': 2}	{'e': 5, 'd': 4, 'f': 6}

	quadratic_features
1	{'b:d, b:d': 16, 'b:d, b:e...': 1}
2	{'b:d, b:d': 16, 'b:d, b:e...': 1}
3	{'b:d, b:d': 16, 'b:d, b:e...': 1}

[3 rows x 3 columns]

Feature Binning

Feature binning is a method of turning continuous variables into categorical values. This is accomplished by grouping the values into a pre-defined number of bins. The continuous value then gets replaced by a string describing the bin that contains that value.

FeatureBinner supports both logarithmic and quantile binning strategies. If the strategy is logarithmic, num_bins is a parameter passed to the constructor and bin break points are defined by 10^{*i} for $i \in [0, \dots, \text{num_bins}]$. For instance, if num_bins = 2, the bins become $(-\text{Inf}, 1]$, $(1, \text{Inf}]$. If num_bins = 3, the bins become $(-\text{Inf}, 1]$, $(1, 10]$, $(10, \text{Inf}]$. If the strategy is quantile, the bin breaks are defined by the num_bins-quantiles for that column's data. Quantiles are values that separate the data into roughly equal-sized subsets.

Introductory Example

```
from graphlab.toolkits.feature_engineering import *

# Construct a feature binner with default options.
sf = graphlab.SFrame({'a': [1,2,3], 'b' : [2,3,4], 'c': [9,10,11]})
binner = graphlab.feature_engineering.create(sf, FeatureBinner())

# Transform the data using the binner.
binned_sf = binner.transform(sf)

# Save the transformer.
binner.save('save-path')

# Bin only a single column 'a'.
binner = graphlab.feature_engineering.create(sf,
                                              FeatureBinner(features = ['a']))

# Bin all columns except 'a'.
binner = graphlab.feature_engineering.create(sf,
                                              FeatureBinner(excluded_features = ['a']))
```

Fitting and transforming

Once a FeatureBinner object is constructed, it must first be fitted and then the transform function can be called to generate hashed features.

Logarithmic strategy:

```
sf = graphlab.SFrame({'a' : range(100), 'b' : range(100)})
binner = graphlab.feature_engineering.FeatureBinner()
fit_binner = binner.fit(sf)
binned_sf = fit_binner.transform(sf)
```

Columns:

Rows: 100

Data:

```

+-----+-----+
|     a     |     b     |
+-----+-----+
| (-Inf, 1] | (-Inf, 1] |
| (-Inf, 1] | (-Inf, 1] |
| (1, 10]   | (1, 10]   |
| (1, 10]   | (1, 10]   |
| (1, 10]   | (1, 10]   |
| (1, 10]   | (1, 10]   |
| (1, 10]   | (1, 10]   |
| (1, 10]   | (1, 10]   |
| (1, 10]   | (1, 10]   |
| (1, 10]   | (1, 10]   |
| (1, 10]   | (1, 10]   |
| (1, 10]   | (1, 10]   |
| (1, 10]   | (1, 10]   |
| (1, 10]   | (1, 10]   |
| (1, 10]   | (1, 10]   |
| (1, 10]   | (1, 10]   |
| (1, 10]   | (1, 10]   |
|     ...    |     ...    |
+-----+-----+
[100 rows x 2 columns]

```

Note: Only the head of the SFrame is printed.

You can use `print_rows(num_rows=m, num_columns=n)` to print more rows and columns.

Quantile strategy

```
sf = graphlab.SFrame({'a' : range(100), 'b' : range(100)})
binner = graphlab.feature_engineering.FeatureBinner(strategy='quantile')
fit_binner = binner.fit(sf)
binned_sf = fit_binner.transform(sf)
```

```
Columns:
    a    str
    b    str

Rows: 100

Data:
+-----+-----+
|     a    |    b    |
+-----+-----+
| (-Inf, 0] | (-Inf, 0] |
| (0, 10]   | (0, 10]  |
| (0, 10]   | (0, 10]  |
| (0, 10]   | (0, 10]  |
| (0, 10]   | (0, 10]  |
| (0, 10]   | (0, 10]  |
| (0, 10]   | (0, 10]  |
| (0, 10]   | (0, 10]  |
| (0, 10]   | (0, 10]  |
| (0, 10]   | (0, 10]  |
| ...       | ...      |
+-----+-----+
[100 rows x 2 columns]
Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.
```

Numeric Imputer

NumericImputer allows you to impute missing values with feature means. Input columns to the NumericImputer must be of type `int`, `float`, `dict`, `list`, or `array.array`. For each input column, the transformed output is a column where the input is retained as-is if:

- there is no missing value.

Inputs that do not satisfy the above are set to the mean value of that feature.

The behavior for different input data column types is as follows:

- **float**: If there is a missing value, it is replaced with the mean of that column.
- **int**: Behaves the same way as `float`.
- **list**: Each index of the list is treated as a feature column, and missing values are replaced with per-feature means. This is the same as unpacking, computing the mean, and re-packing. See [pack_columns](#) for more information. All elements must be of type `float`, `int`, or `None`.
- **array**: Same behavior as `list`
- **dict** : Same behavior as `list`, except keys not present in a particular row are implicitly interpreted as having the value 0. This makes the `dict` type a sparse representation of a vector.

Introductory Example

```
# Create data.
sf = graphlab.SFrame({'a': [1, None, 3],
                      'b' : [2, None, 4]})

# Create a transformer.
from graphlab.toolkits.feature_engineering import NumericImputer
imputer = graphlab.feature_engineering.create(sf, NumericImputer())

# Transform the data.
transformed_sf = imputer.transform(sf)

# Save the transformer.
imputer.save('save-path')

# Return the means.
imputer['means']
```

Columns:

a	float
b	float

Rows: 1

Data:

a	b
2.0	3.0

[1 rows x 2 columns]

Fitting and transforming

```
# Integer/Float columns
# -----
# Create the data
sf = graphlab.SFrame({'a' : [1, 2, None, 4, 5],
                      'b' : [2, 3, None, 5, 6]})

# Create the imputer.
imputer = graphlab.feature_engineering.NumericImputer()

# Fit and transform on the same data.
transformed_sf = imputer.fit_transform(sf)
```

```
Columns:
  a    float
  b    float
```

Rows: 5

Data:

```
+----+----+
|  a |  b |
+----+----+
| 1.0 | 2.0 |
| 2.0 | 3.0 |
| 3.0 | 4.0 |
| 4.0 | 5.0 |
| 5.0 | 6.0 |
+----+----+
[5 rows x 2 columns]
```

Lists can contain numeric and None values.

```
sf = graphlab.SFrame({'a': [[1, 2],
                            [2, 3],
                            [3, 4],
                            [None, None],
                            [5, 6],
                            [6, 7]])}

# Construct and fit.
from graphlab.toolkits.feature_engineering import NumericImputer
imputer = graphlab.feature_engineering.create(sf, NumericImputer())

# Transform the data
transformed_sf = imputer.transform(sf)
```

```
Columns:
    a    list
```

Rows: 6

Data:

```
+-----+
|   a    |
+-----+
| [1, 2] |
| [2, 3] |
| [3, 4] |
| [3.4, 4.4] |
| [5, 6] |
| [6, 7] |
+-----+
[6 rows x 1 columns]
```

Dictionaries can contain numeric and None values. Assumes sparse data format.

```
sf = graphlab.SFrame({'X':
                      [{a:1, b: 2, c: 3},
                       None,
                       {b:4, c: None, d: 6}]}))

# Construct and fit.
from graphlab.feature_engineering import NumericImputer
imputer = graphlab.feature_engineering.create(sf, NumericImputer())

# Transform the data
transformed_sf = imputer.transform(sf)
```

```
Columns:
    X    dict
```

Rows: 3

Data:

```
+-----+
|           X           |
+-----+
| {'a': 1, 'c': 3, 'b': 2} |
| {'a': 0.5, 'c': 3.0, 'b': ...} |
| {'c': 3.0, 'b': 4, 'd': 6} |
+-----+
[3 rows x 1 columns]
```


Categorical features

These feature transformations are useful when you have categorical data, i.e., data where each observation is one of a discrete set of possible values.

- [One Hot Encoder](#)
- [Count Thresholder](#)
- [Categorical Imputer](#)
- [Count Featurizer](#)

One-Hot-Encoder

Encode a collection of categorical features using a 1-of-K encoding scheme. Input columns to the one-hot-encoder must be of type int, string, dict, or list. The transformed output is a column of type dictionary (max_categories per column dimension sparse vector) where the key corresponds to the index of the categorical variable and the value is 1.

The behaviour of the one-hot-encoder for each input data column type is as follows (see transform() for examples of the same):

- **string** : The key in the output dictionary is the string category and the value is 1.
- **int** : Behave similar to string columns.
- **list** : Each value in the list is treated like an individual string. Hence, a list of categorical variables can be used to represent a feature where all categories in the list are simultaneously hot.
- **dict** : They key of the dictionary is treated as a namespace and the value is treated as a sub-category in the namespace. The categorical variable being encoded in this case is a combination of the namespace and the sub-category.

You can specify the number of categories with the parameter max_categories.

Introductory Example

```
# Create data.  
sf = graphlab.SFrame({'a': [1,2,3], 'b' : [2,3,4]})  
  
# Create a one-hot encoder.  
from graphlab.toolkits.feature_engineering import OneHotEncoder  
encoder = graphlab.feature_engineering.create(sf, OneHotEncoder())  
  
# Transform the data.  
transformed_sf = encoder.transform(sf)
```

```
Columns:
  encoded_features    dict

Rows: 5

Data:
+-----+
| encoded_features |
+-----+
| {0: 1, 1: 1, 2: 1} |
| {2: 1, 3: 1, 4: 1} |
| {5: 1, 6: 1, 7: 1} |
| {2: 1, 3: 1, 4: 1} |
| {0: 1, 3: 1, 6: 1} |
+-----+
[5 rows x 1 columns]
```

```
# Save the transformer.
encoder.save('save-path')

# Return the indices in the encoding.
encoder['feature_encoding']
```

```
Columns:
  feature    str
  category   str
  index      int

Rows: 4

Data:
+-----+-----+-----+
| feature | category | index |
+-----+-----+-----+
| a      | 2       | 0     |
| a      | 3       | 1     |
| b      | 2       | 2     |
| b      | 3       | 3     |
+-----+-----+-----+
[4 rows x 3 columns]
```

Fitting and transforming

Once a OneHotEncoder object is constructed, it must first be fitted and then the transform function can be called to generate encoded features.

```
# String/Integer columns
#
from graphlab.toolkits.feature_engineering import OneHotEncoder
sf = graphlab.SFrame({'a' : [1,2,3,2,3], 'b' : [2,3,4,2,3]})

# Create a OneHotEncoder
encoder = graphlab.feature_engineering.create(sf, OneHotEncoder())

# Fit and transform on the same data.
transformed_sf = encoder.fit_transform(sf)
```

Columns:
encoded_features dict

Rows: 5

Data:

encoded_features
{0: 1, 3: 1}
{1: 1, 4: 1}
{2: 1, 5: 1}
{1: 1, 3: 1}
{2: 1, 4: 1}

[5 rows x 1 columns]

```
# Lists can be used to encode sets of categories for each example.
#
from graphlab.toolkits.feature_engineering import OneHotEncoder
sf = graphlab.SFrame({'categories': [['cat', 'mammal'],
                                      ['dog', 'mammal'],
                                      ['human', 'mammal'],
                                      ['seahawk', 'bird'],
                                      ['wasp', 'insect']]})

# Construct and fit.
encoder = graphlab.feature_engineering.create(sf, OneHotEncoder())

# Transform the data
transformed_sf = encoder.transform(sf)
```

```
Columns:
  encoded_features    dict
```

Rows: 5

Data:

encoded_features
{0: 1, 1: 1}
{0: 1, 2: 1}
{0: 1, 3: 1}
{4: 1, 6: 1}
{5: 1, 7: 1}

[5 rows x 1 columns]

```
# Dictionaries can be used for name spaces & sub-categories.
# -----
from graphlab.toolkits.feature_engineering import OneHotEncoder
sf = graphlab.SFrame({'attributes':
    [{"height':'tall', 'age': 'senior', 'weight': 'thin'},
     {"height':'short', 'age': 'child', 'weight': 'thin'},
     {"height':'giant', 'age': 'adult', 'weight': 'fat'},
     {"height':'short', 'age': 'child', 'weight': 'thin'},
     {"height':'tall', 'age': 'child', 'weight': 'fat"}]})

# Construct and fit.
encoder = graphlab.feature_engineering.create(sf, OneHotEncoder())

# Transform the data
transformed_sf = encoder.transform(sf)
```

```
Columns:
  encoded_features    dict
```

Rows: 5

Data:

encoded_features
{0: 1, 1: 1, 2: 1}
{2: 1, 3: 1, 4: 1}
{5: 1, 6: 1, 7: 1}
{2: 1, 3: 1, 4: 1}
{0: 1, 3: 1, 6: 1}

[5 rows x 1 columns]

Count Thresholdolder

Count Thresholdolder allows you to map infrequent categorical variables to a new/separate category. Input columns to the CountThresholdolder must by of type **string**, **int**, **list**, or **dict**. For each column in the input, the transformed output is a column where the input category is retained as-is if it has occurred at least threshold times in the training data. Categories that do not satisfy the above are set to `output_category_name`.

The behaviour for different input data column types is as follows: (see `transform()` for examples).

- **string** : Strings are marked with the `output_category_name` if the threshold condition described above is not satisfied.
- **int** : Behave the same way as string. If `output_category_name` is of type string, then the entire column is cast to string.
- **list** : Each of the values in the list are mapped in the same way as a string value.
- **dict** : They key of the dictionary is treated as a namespace and the value is treated as a sub-category in the namespace. The categorical variable passed through the transformer is a combination of the namespace and the sub-category.

You specify the threshold at which to preserve the categories with the parameter "threshold".

Introductory Example

```
from graphlab.toolkits.feature_engineering import *

# Create data.
sf = gl.SFrame({'a': [1,2,3], 'b' : [2,3,4]})

# Create a transformer.
count_tr = gl.feature_engineering.create(sf, CountThresholdolder(threshold = 1))

# Transform the data.
transformed_sf = count_tr.transform(sf)

# Save the transformer.
count_tr.save('save-path')

# Return the categories that are not discarded.
count_tr['categories']
```

```
Columns:
  feature str
  category str
```

Rows: 6

Data:

```
+-----+-----+
| feature | category |
+-----+-----+
|   a     |    1    |
|   a     |    2    |
|   a     |    3    |
|   b     |    2    |
|   b     |    3    |
|   b     |    4    |
+-----+-----+
[6 rows x 2 columns]
```

Fitting and transforming

Once a CountThreshold object is constructed, it must first be fitted and then the transform function can be called to generate encoded features.

```
# String/Integer columns
# -----
sf = gl.SFrame({'a' : [1,2,3,2,3], 'b' : [2,3,4,2,3]})

# Set all categories that did not occur at least 2 times to None.
count_tr = gl.feature_engineering.CountThreshold(threshold = 2)

# Fit and transform on the same data.
transformed_sf = count_tr.fit_transform(sf)
```

```
Columns:
a    int
b    int

Rows: 3

Data:
+-----+-----+
|   a   |   b   |
+-----+-----+
| None |   2   |
|   2   |   3   |
|   3   | None  |
|   2   |   2   |
|   3   |   3   |
+-----+
[5 rows x 2 columns]
```

```
# Lists can be used to encode sets of categories for each example.
# -----
sf = gl.SFrame({'categories': [['cat', 'mammal'],
                               ['cat', 'mammal'],
                               ['human', 'mammal'],
                               ['seahawk', 'bird'],
                               ['duck', 'bird'],
                               ['seahawk', 'bird']]})

# Construct and fit.
from graphlab.toolkits.feature_engineering import CountThreshold
count_tr = graphlab.feature_engineering.create(sf, CountThreshold(threshold = 2))

# Transform the data
transformed_sf = count_tr.transform(sf)
```

```
Columns:
    categories  list
```

Rows: 6

Data:

categories	list
[cat, mammal]	
[cat, mammal]	
[None, mammal]	
[seahawk, bird]	
[None, bird]	
[seahawk, bird]	

[6 rows x 1 columns]

```
# Dictionaries can be used for name spaces & sub-categories.
```

```
# -----
sf = gl.SFrame({'attributes':
                 [{"height':'tall', 'age': 'senior', 'weight': 'thin'},
                  {"height':'short', 'age': 'child', 'weight': 'thin'},
                  {"height':'giant', 'age': 'adult', 'weight': 'fat'},
                  {"height':'short', 'age': 'child', 'weight': 'thin'},
                  {"height':'tall', 'age': 'child', 'weight': 'fat'}]})

# Construct and fit.
from graphlab.toolkits.feature_engineering import CountThreshold
count_tr = gl.feature_engineering.create(sf,
                                         CountThreshold(threshold = 2))

# Transform the data
transformed_sf = count_tr.transform(sf)
```

```
Columns:
    attributes      dict

Rows: 5

Data:
+-----+
|       attributes      |
+-----+
| {'age': None, 'weight': 't... | 
| {'age': 'child', 'weight':... | 
| {'age': None, 'weight': No... | 
| {'age': 'child', 'weight':... | 
| {'age': 'child', 'weight':... | 
+-----+
```

Categorical Imputer

Impute missing categorical values using reference features. The imputer takes as an input a column which contains categorical values, and may contain some None values. Its secondary input is a set of reference feature columns. During the Fit phase, the imputer learns the relationships between the values of the column to be imputed and the reference feature columns. During the Transform phase, the learned relationship is applied to the missing values, and they are filled with categorical values. The imputer also outputs a probability for each filled value. The probability is based on the number of possible candidates for each missing value.

In specific terms, during the Fit phase, the imputer performs two steps. The first step is to cluster the data using the reference features. The dominant label of the cluster is used to provide a label to None values. The probability returned becomes the proportion of dominant label in the cluster. The second step involves clusters with no label. In this case, a graph of clusters is built, and label propagation is used to determine the best label for the cluster. In this case, the probability returned is established by the label propagation algorithm.

The reference feature columns must be of type *int*, *float*, *dict*, *list*, *string* or *array.array*. The column to be imputed must be of the type *int*, *dict*, *list*, *string* or *array.array* - as long as it is categorical.

The column to be imputed can contain Nones during the Fit phase, but it must also contain some categorical values. In the Transform phase, it can be all None, or can contain some values. If it contains values, they will be returned untouched with a probability of 100%.

Introductory Example

```
# Import GraphLab if not already imported
import graphlab

# Create data.
sf = graphlab.SFrame({'a': [1,0,1], 'b' : [0,1,0], 'c' : ['a', 'b', None]})

# Create a transformer that fits learns from the data above, and tries to impute column c
from graphlab.toolkits.feature_engineering import CategoricalImputer
imputer = graphlab.feature_engineering.create(sf, CategoricalImputer(feature = 'c'))

# Transform the data.
transformed_sf = imputer.transform(sf)

# Retrieve the imputed values
transformed_sf
```

Columns:

a	int
b	int
c	str
predicted_feature_c	str
feature_probability_c	float

Rows: 3

Data:

a	b	c	predicted_feature_c	feature_probability_c
1	0	a	a	1.0
0	1	b	b	1.0
1	0	None	a	1.0

[3 rows x 5 columns]

Fitting and transforming

```
# Import GraphLab if not already imported
import graphlab

# Fit and Transform the same column
# -----
# Create the data
sf = graphlab.SFrame({'a': [1,0,1], 'b' : [0,1,0], 'c' : ['a', 'b', None]})

# Create the imputer.
imputer = graphlab.feature_engineering.CategoricalImputer(feature='c')

# Fit and transform on the same data.
transformed_sf = imputer.fit_transform(sf)

#Retrieve the imputed values
transformed_sf
```

Columns:

```
a    int
b    int
c    str
predicted_feature_c    str
feature_probability_c   float
```

Rows: 3

Data:

a	b	c	predicted_feature_c	feature_probability_c
1	0	a	a	1.0
0	1	b	b	1.0
1	0	None	a	1.0

[3 rows x 5 columns]

```
# Import GraphLab if not already imported
import graphlab

# Fit on one set, and transform another
#
sf = graphlab.SFrame({'a': [1,0,1], 'b' : [0,1,1], 'c' : ['a', 'b', 'c']})

# Construct and fit.
from graphlab.toolkits.feature_engineering import CategoricalImputer
imputer = graphlab.feature_engineering.CategoricalImputer(feature='c')

# Fit the data
imputer.fit(sf)
```

Class : CategoricalImputer

Model fields

reference_features : ['a', 'b']
Column to impute : c

```
# Data to be imputed
sf2 = graphlab.SFrame({'a': [1,0,1,0], 'b' : [0,1,1,0], 'c' : [None, None, None, None]})  
sf2['c'] = sf2['c'].astype(str)

# Transform the data
transformed_sf = imputer.transform(sf2)

#Retrieve the imputed values
transformed_sf
```

```
Columns:  
a    int  
b    int  
c    float  
predicted_feature_c    str  
feature_probability_c    float
```

Rows: 4

Data:

a	b	c	predicted_feature_c	feature_probability_c
1	0	None	a	1.0
0	1	None	c	0.5
1	1	None	c	0.5
0	0	None	c	0.5

[4 rows x 5 columns]

Count Featurizer

The Count Featurizer (Also, called Learning With Counts, or Dracula

<https://blogs.technet.microsoft.com/machinelearning/2015/02/17/big-learning-made-easy-with-counts/>) allows you to convert complex categorical dimensions to simpler scalar dimensions which are easier and faster to train on while improving classification performance.

The Count Featurizer is unlike most of the other feature engineering methods in that, it is designed specifically for classification, and requires some care to use correctly.

- The count featurizer requires a target prediction column to be provided.
- You should perform fit and transform on different datasets to avoid overfitting (See Usage Tips below).

Formally, given a target column Y we are trying to predict with k classes ($1 \dots k$), it replaces every categorical column X with 2 columns:

count_X : a list of the following values

- $\#(Y = 1 \ \& \ X = x_i)$: the number of times $Y = 1$ when X has the value x_i
- $\#(Y = 2 \ \& \ X = x_i)$: the number of times $Y = 2$ when X has the value x_i
- $\#(Y = 3 \ \& \ X = x_i)$: the number of times $Y = 3$ when X has the value x_i
- ...
- $\#(Y = k \ \& \ X = x_i)$: the number of times $Y = k$ when X has the value x_i

prob_X : a list of the following values

- $P(Y = 1 \mid X = x_i)$: the probability $Y = 1$ when X has the value x_i
- $P(Y = 2 \mid X = x_i)$: the probability $Y = 2$ when X has the value x_i
- $P(Y = 3 \mid X = x_i)$: the probability $Y = 3$ when X has the value x_i
- ...
- $P(Y = k-1 \mid X = x_i)$: the probability $Y = k-1$ when X has the value x_i

The input categorical columns must of type **string** or **int** and the target prediction column must also be of type **string** or **int**.

Usage

```
import graphlab
from graphlab.toolkits.feature_engineering import *

# Create data.
sf=graphlab.SFrame({'state':[0,1,0,3,2,2],
                    'gender':['M','F','M','M','F','F'],
                    'click':[1,1,0,1,1,1]})

# Split the data
sf_fit = sf[:4]
sf_train = sf[4:]

# Create a transformer.
countfeat = graphlab.feature_engineering.create(sf_fit,
                                                 CountFeaturizer(target='click'))

# Transform the train set. This is the dataset I will train my classifier on
transformed_sf_train = countfeat.transform(sf_train)

# Save the transformer.
countfeat.save('save-path')

sf_train
transformed_sf_train
```

```

Columns:
  click      int
  gender     str
  state      int

Rows: 2

Data:
+-----+-----+-----+
| click | gender | state |
+-----+-----+-----+
|   1   |    F   |    2   |
|   1   |    F   |    2   |
+-----+-----+-----+
[2 rows x 3 columns]

Columns:
  count_gender    array
  prob_gender    array
  count_state    array
  prob_state     array
  click         int

Rows: 2

Data:
+-----+-----+-----+-----+-----+
| count_gender | prob_gender | count_state | prob_state | click |
+-----+-----+-----+-----+-----+
| [0.0, 1.0]  |    [0.0]    | [0.0, 0.0] |    [0.0]    |    1    |
| [0.0, 1.0]  |    [0.0]    | [0.0, 0.0] |    [0.0]    |    1    |
+-----+-----+-----+-----+-----+
[2 rows x 5 columns]

```

Usage Tips

Since the Count Featurizer internally learns something similar to a Naive Bayes classifier you should perform fit and transform on different datasets to avoid overfitting.

Entire Dataset

Train	Validation
-------	------------

Fit	Transform+Train	Validation
-----	-----------------	------------

Furthermore, if your data has a temporal component to it (for instance log data for click through prediction), you should not perform a random split, but perform the split temporally: the fit dataset should be the oldest, the validation should be the newest, and the training set in between.

Text features

These feature transformations are useful when you have text data.

- [TF-IDF](#)
- [Tokenizer](#)
- [BM25](#)

TF-IDF

The prototypical application of TF-IDF transformations involves document collections, where each element represents a document. Documents are represented in a bag-of-words format, i.e. a dictionary whose keys are words and whose values are the number of times the word occurs in the document. For more details and further reading, check the reference section.

The TF-IDF transformation performs the following computation

$$\text{TF-IDF}(w, d) = tf(w, d) * \log(N/f(w))$$

where $tf(w, d)$ is the number of times word w appeared in document d , $f(w)$ is the number of documents word w appeared in, N is the number of documents, and we use the natural logarithm.

The transformed output is a column of type dictionary (`max_categories` per column dimension sparse vector) where the key corresponds to the index of the categorical variable and the value is `1`.

The behavior of TF-IDF for each input data column type for supported types is as follows:

- **dict**: Each (key, value) pair is treated as count associated with the key for this row. A common example is to have a dict element contain a bag-of-words representation of a document, where each key is a word and each value is the number of times that word occurs in the document. All non-numeric values are ignored.
- **list**: The list is converted to a bag of words format, where the keys are the unique elements in the list and the values are the counts of those unique elements. After this step, the behaviour is identical to dict.
- **string**: Behaves identically to a **dict**, where the dictionary is generated by converting the string into a bag-of-words format. For example, 'I really like really fluffy dogs' would get converted to {'I': 1, 'really': 2, 'like': 1, 'fluffy': 1, 'dogs': 1}.

Introductory Example

```
import graphlab as gl

# Create data.
sf = gl.SFrame({'a': ['1','2','3'], 'b' : [2,3,4]})

# Create a one-hot encoder.
from graphlab.toolkits.feature_engineering import TFIDF
encoder = gl.feature_engineering.create(sf, TFIDF('a'))

# Transform the data.
transformed_sf = encoder.transform(sf)
```

```
Columns:
    a dict
    b int

Rows: 3

Data:
+-----+---+
|      a      | b |
+-----+---+
| {'1': 1.0986122886681098} | 2 |
| {'2': 1.0986122886681098} | 3 |
| {'3': 1.0986122886681098} | 4 |
+-----+---+
[3 rows x 2 columns]
```

```
# Save the transformer.
>>> encoder.save('save-path')

# Return the indices in the encoding.
>>> encoder['document_frequencies']
```

```

Columns:
  feature_column  str
  term      str
  document_frequency  str

Rows: 3

Data:
+-----+-----+-----+
| feature_column | term | document_frequency |
+-----+-----+-----+
|      a        | 1    |          1           |
|      a        | 2    |          1           |
|      a        | 3    |          1           |
+-----+-----+-----+
[3 rows x 3 columns]

```

Fitting and Transforming

```

# For list columns:

l1 = ['a','good','example']
l2 = ['a','better','example']
sf = gl.SFrame({'a' : [l1,l2]})
tfidf = gl.feature_engineering.TFIDF('a')
fit_tfidf = tfidf.fit(sf)
transformed_sf = fit_tfidf.transform(sf)

```

```

Columns:
  a    dict

Rows: 2

Data:
+-----+
|      a        |
+-----+
| {'a': 0.0, 'good': 0.69314... |
| {'better': 0.6931471805599... |
+-----+
[2 rows x 1 columns]

```

```
# For string columns:

sf = gl.SFrame({'a' : ['a good example', 'a better example']})
tfidf = gl.feature_engineering.TFIDF('a')
fit_tfidf = tfidf.fit(sf)
transformed_sf = fit_tfidf.transform(sf)
```

```
Columns:
    a    dict

Rows: 2

Data:
+-----+
|      a      |
+-----+
| {'a': 0.0, 'good': 0.69314... |
| {'better': 0.6931471805599... |
+-----+
[2 rows x 1 columns]
```

```
# For dictionary columns:
sf = gl.SFrame(
    {'docs': [{['this': 1, 'is': 1, 'a': 2, 'sample': 1},
              {'this': 1, 'is': 1, 'another': 2, 'example': 3}]})
tfidf = gl.feature_engineering.TFIDF('docs')
fit_tfidf = tfidf.fit(sf)
transformed_sf = fit_tfidf.transform(sf)
```

```
Columns:
    docs    dict

Rows: 2

Data:
+-----+
|      docs      |
+-----+
| {'this': 0.0, 'a': 1.38629... |
| {'this': 0.0, 'is': 0.0, '... |
+-----+
[2 rows x 1 columns]
```

Tokenizer

For an SFrame of strings, where each row is assumed to be a natural English language document, the tokenizer transforms each row into an ordered list of strings that represents a simpler version of the Penn-Tree-Bank-style (PTB-style) tokenization of that row's document. For many text analytics tasks that require word-level granularity, simple space delimitation does not address some of the subtleties of natural language text, especially with respect to sentence-final punctuation, contractions, URL's, email addresses, phone numbers and other quirks. The representation of a document provided by PTB-style tokenization is essential for sequence-tagging, parsing, bag-of-words treatment, and any text analytics task that requires word-level granularity. For a description of this style of tokenization, see <https://www.cis.upenn.edu/~treebank/tokenization.html>. Note that our tokenizer does not normalize quote and bracket-like characters as described by the linked document.

The transformed output is a column of type list[string] with the list of tokens for each document.

```
import graphlab as gl

sf = gl.SFrame({'docs': ["This is a document!",
                        "This one's also a document."]})
tokenizer = graphlab.feature_engineering.Tokenizer(features = ['docs'])
tokenizer.fit(sf)
tokenized_sf = tokenizer.transform(sf)
```

```
Data:
+-----+
|      docs      |
+-----+
| ['This', 'is', 'a', 'docum... | 
| ['This', 'one', '\'s', 'al... | 
+-----+
[2 rows x 1 columns]
```

For the tokenizer, the transformation is completely dependent on internal state prior to seeing any data, so it doesn't need to be serialized or fit.

We encourage you to use the 'tokenize' function in the text_analytics toolkit. It has nearly identical behavior to Tokenizer's transform function, but takes only the target SArray instead of an SFrame.

Rare Word Trimmer

Removing words that occur below a certain number of times in a given column is a common method of cleaning text before it is used, and can increase the quality and explainability of the models learned on the transformed data. For instance, rare words are generally given higher weight in a [TF-IDF](#) transform. However, the rarest words are frequently misspellings of common words. Interpreting these words as informative is problematic in the modeling stage of analysis.

`RareWordTrimmer` can be applied to all the string-, dictionary-, and list-typed columns in a given SFrame. Its behavior for each supported input column type is as follows.

- **string** : The string is first tokenized. By default, all letters are first converted to lower case, then tokenized by space characters. Each token is taken to be a word, and words occurring below a threshold number of times across the entire column are removed, then the remaining tokens are concatenated back into a string.
- **list** : Each element of the list must be a string, where each element is assumed to be a token. The remaining tokens are then filtered by count occurrences and a threshold value.
- **dict** : The method first obtains the list of keys in the dictionary. This list is then processed as a standard list, except the value of each key must be of integer type and is considered to be the count of that key.

Notes

If the SFrame to be transformed already contains a column with the designated output column name, then that column will be replaced with the new output. In particular, this means that `output_column_prefix=None` will overwrite the original feature columns.

The output of the [Tokenizer](#) object produces valid `list` input for this method, and the output of the [WordCounter](#) object produces valid `dict` input.

References

- [Penn treebank tokenization] (<https://www.cis.upenn.edu/~treebank/tokenization.html>)

Introductory examples

```

import graphlab as gl

# Create data.
sf = gl.SFrame({
...    'string': ['sentences Sentences', 'another sentence another year'],
...    'dict': [{bob: 1, Bob: 2}, {a: 0, cat: 5}],
...    'list': [['one', 'two', 'three', 'Three'], [a, cat, Cat]]})

# Create a RareWordTrimmer transformer.
from graphlab.toolkits.feature_engineering import RareWordTrimmer
trimmer = RareWordTrimmer(threshold=2)

# Fit and transform the data.
transformed_sf = trimmer.fit_transform(sf)

```

Columns:

dict	dict
list	list
string	str

Rows: 2

Data:

dict	list	string
{'bob': 2}	[three, three]	sentences sentences
{'cat': 5}	[cat, cat]	another another

[2 rows x 3 columns]

```

# Save the transformer.
trimmer.save('save-path')

```

Fit and transform

```

import graphlab as gl

# For list columns (string elements converted to lower case by default):

l1 = ['a', 'good', 'example']
l2 = ['a', 'better', 'example']
sf = gl.SFrame({'a' : [l1,l2]})

wt = gl.feature_engineering.RareWordTrimmer('a', threshold=2)
fit_wt = wt.fit(sf)
transformed_sf = fit_wt.transform(sf)

```

Columns:
a list

Rows: 2

Data:

a
[a, example]
[a, example]

[2 rows x 1 columns]

```
# For string columns (converted to lower case by default):
```

```

sf = gl.SFrame({'a' : ['a good example', 'a better example']})
wc = gl.feature_engineering.RareWordTrimmer('a', threshold=2)
fit_wt = wt.fit(sf)
transformed_sf = fit_wt.transform(sf)

```

Columns:
a str

Rows: 2

Data:

a
a example
a example

[2 rows x 1 columns]

```
# For dictionary columns (keys converted to lower case by default):
sf = gl.SFrame(
...     {'docs': [{['this': 1, 'is': 1, 'a': 2, 'sample': 1},
...               {'this': 1, 'IS': 1, 'another': 2, 'example': 3}]}})
wt = gl.feature_engineering.RareWordTrimmer('docs', threshold=2)
fit_wt = wt.fit(sf)
transformed_sf = fit_wt.transform(sf)
```

```
Columns:
docs      dict
```

```
Rows: 2
```

```
Data:
```

```
+-----+
|       docs       |
+-----+
|  {'this': 1, 'a': 2, 'is': 1} |
|  {'this': 1, 'is': 1, 'exam... |
+-----+
[2 rows x 1 columns]
```

BM25

The BM25 function scores each document in a corpus according to the document's relevance to a particular text query. For a query with terms q_1, \dots, q_n , the BM25 score for document d is:

$$\text{BM25}(d) = \sum_{i=1}^n \text{IDF}(q_i) \frac{f(q_i) * (k_1 + 1)}{f(q_i) + k_1 * (1 - b + b * |D|/d_{avg})}$$

where:

- $f(q_i)$ is the number of times term q_i occurs in document d ,
- $|D|$ is the number of words in document d ,
- d_{avg} is the average number of words per document,
- b and k_1 are free parameters for Okapi BM25,

The first quantity in the sum is the inverse document frequency. For a corpus with N documents, inverse document frequency for term q_i is:

$$\text{IDF}(q_i) = \log \frac{N - N(q_i) + 0.5}{N(q_i) + 0.5}$$

where $N(q_i)$ is the number of documents in the corpus that contain term q_i .

The transformed output is a column of type float with the BM25 score for each document. For more details on the BM25 score see http://en.wikipedia.org/wiki/Okapi_BM25.

The behavior of BM25 for different input data column types is as follows:

- **dict** : Each (key, value) pair is treated as count associated with the key for this row. A common example is to have a dict element contain a bag-of-words representation of a document, where each key is a word and each value is the number of times that word occurs in the document. All non-numeric values are ignored.
- **list** : The list is converted to bag of words of format, where the keys are the unique elements in the list and the values are the counts of those unique elements. After this step, the behaviour is identical to dict.
- **string** : Behaves identically to a **dict**, where the dictionary is generated by converting the string into a bag-of-words format. For example, "I really like really fluffy dogs" would get converted to `{'I' : 1, 'really': 2, 'like': 1, 'fluffy': 1, 'dogs':1}` .

Introductory Example

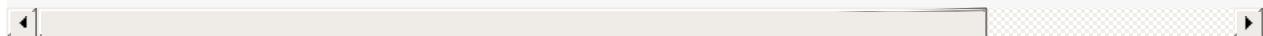
```
import graphlab as gl

# Create data.
sf = gl.SFrame({'docs': [{"this': 1, 'is': 1, 'a': 2, 'sample': 1},
                           {"this": 1, 'is': 1, 'another': 2, 'example': 3},
                           {"final": 1, 'doc': 1, 'here': 2}]}))

# Create a query set
query = ['a', 'query', 'example']

# Create a BM25 encoder
from graphlab.toolkits.feature_engineering import BM25
encoder = gl.feature_engineering.create(dataset = sf, transformers = BM25(feature = 'docs'))

# Transform the data.
transformed_sf = encoder.transform(sf)
```



```
Data:
+-----+
|      docs      |
+-----+
| 0.744711615513 |
| 0.789682123696 |
|      0.0       |
+-----+
[3 rows x 1 columns]
```

```
# Save the transformer.
encoder.save('save-path')

# Return the indices in the encoding.
encoder['document_frequencies']
```

```
Data:
+-----+-----+-----+
| feature_column | term   | document_frequency |
+-----+-----+-----+
|      docs      |    a    |          1          |
|      docs      | example |          1          |
+-----+-----+-----+
[2 rows x 3 columns]
```

Fitting and Transforming

```

import graphlab as gl

# Dictionary Input:
sf = gl.SFrame({'docs': [{"this': 1, 'is': 1, 'a': 2, 'sample': 1},
                           {"this": 1, 'is': 1, 'another': 2, 'example': 3},
                           {"final": 1, 'doc': 1, 'here': 2}]})

# Create a query set
query = ['a', 'query', 'example']
encoder = gl.feature_engineering.BM25(feature = 'docs', query = query)
encoder = encoder.fit(data = sf)
transformed_sf = encoder.transform(data = sf)
transformed_sf

```

```

+-----+
|      docs      |
+-----+
| 0.744711615513 |
| 0.789682123696 |
|      0.0       |
+-----+
[3 rows x 1 columns]

```

```

# List Input:
l1 = ['this', 'is', 'a', 'a', 'sample']
l2 = ['this', 'is', 'another', 'another', 'example', 'example', 'example']
l3 = ['final', 'doc', 'here', 'here']
sf = gl.SFrame({'docs' : [l1,l2,l3]})

# Create a query set
query = ['a', 'query', 'example']
encoder = gl.feature_engineering.BM25(feature = 'docs', query = query)
encoder = encoder.fit(data = sf)
transformed_sf = encoder.transform(data = sf)
transformed_sf

```

```

+-----+
|      docs      |
+-----+
| 0.744711615513 |
| 0.789682123696 |
|      0.0       |
+-----+
[3 rows x 1 columns]

```

```
# String Input:  
s1 = 'this is a a sample'  
s2 = 'this is another another example example example'  
s3 = 'final doc here here'  
sf = gl.SFrame({'docs' : [s1,s2,s3]})  
  
# Create a query set  
query = ['a', 'query', 'example']  
encoder = gl.feature_engineering.BM25(feature = 'docs', query = query)  
encoder = encoder.fit(data = sf)  
transformed_sf = encoder.transform(data = sf)  
transformed_sf
```

```
+-----+  
| docs |  
+-----+  
| 0.744711615513 |  
| 0.789682123696 |  
| 0.0 |  
+-----+[3 rows x 1 columns]
```

Part of Speech Extractor

PartOfSpeechExtractor takes SFrame columns of type string and list, and transforms into a nested dictionary. If the input column is of type list, each element in the list must also be of type list or string. In the first level of the output dictionary, the keys are parts of speech and values are bags of words of the specified part of speech.

Notes

This extractor depends on spaCy, a Python package for natural language processing. Please see spacy.io for installation information.

If the SFrame to be transformed already contains a column with the designated output column name, then that column will be replaced with the new output. In particular, this means that `output_column_prefix=None` will overwrite the original feature columns.

Introductory examples

```
>>> import graphlab as gl

# Create data.
>>> sf = gl.SFrame({
...     'text': ['This is a great sentence. This is sentence two.'])

# Create a PartOfSpeechExtractor transformer.
>>> from graphlab.toolkits.feature_engineering import PartOfSpeechExtractor
>>> transformer = PartOfSpeechExtractor()

# Fit and transform the data.
>>> transformed_sf = transformer.fit_transform(sf)
```

```
Columns:
    text      dict

Rows: 1

Data:
+-----+
|       text       |
+-----+
| {'ADJ': {'great': 1}} |
+-----+
[1 rows x 1 columns]
```

```
#SFrame with list of strings
>>> sf = gl.SFrame({
...     'text': [['This is a great sentence.', 'This is sentence two.']]})

# Create a PartOfSpeechExtractor transformer.
>>> from graphlab.toolkits.feature_engineering import PartOfSpeechExtractor
>>> transformer = PartOfSpeechExtractor()

# Fit and transform the data.
>>> transformed_sf = transformer.fit_transform(sf)
```

```
Columns:
    text      dict

Rows: 1

Data:
+-----+
|      text      |
+-----+
| {'ADJ': {'great': 1}} |
+-----+
[1 rows x 1 columns]
```

```
#SFrame with list of strings
>>> sf = gl.SFrame({
...     'text': [['This is a great sentence.', 'This is sentence two.']]}

# Create a PartOfSpeechExtractor transformer.
>>> from graphlab.toolkits.feature_engineering import PartOfSpeechExtractor
>>> transformer = PartOfSpeechExtractor(
...                         chosen_pos=[graphlab.text_analytics.PartOfSpeech.NOUN,
...                                     graphlab.text_analytics.PartOfSpeech.VERB])

# Fit and transform the data.
>>> transformed_sf = transformer.fit_transform(sf)
```

```
Columns:
  string  dict

Rows: 1

Data:
+-----+
|      string      |
+-----+
| {'NOUN': {'sentence': 1}} |
+-----+
[1 rows x 1 columns]
```

Sentence Splitter

The SentenceSplitter takes SFrame columns of type string or list, and transforms into list of strings, where each element is a single sentence. If the input column type is list, each element must either be list or string and the lists are recursively flattened and concatenated before sentence splitting.

Notes

This transformer depends on spaCy, a Python package for natural language processing. Please see [spacy.io](#) for installation information.

If the SFrame to be transformed already contains a column with the designated output column name, then that column will be replaced with the new output. In particular, this means that `output_column_prefix=None` will overwrite the original feature columns.

Introductory examples

```
>>> import graphlab as gl

# Create data.
>>> sf = gl.SFrame({
...     'text': ['This is sentence 1. This is sentence two.'])

# Create a SentenceSplitter transformer.
>>> from graphlab.toolkits.feature_engineering import SentenceSplitter
>>> transformer = SentenceSplitter()

# Fit and transform the data.
>>> transformed_sf = transformer.fit_transform(sf)
```

```
Columns:
text    list

Rows: 1

Data:
+-----+
|      text      |
+-----+
| [This is sentence 1., This... |
+-----+
[1 rows x 1 columns]
```

```
# For SFrame of type list
>>> import graphlab as gl

# Create data.
>>> sf = gl.SFrame({
...     'text': [['This is sentence 1. This is sentence two.']]})

# Create a SentenceSplitter transformer.
>>> from graphlab.toolkits.feature_engineering import SentenceSplitter
>>> transformer = SentenceSplitter()

# Fit and transform the data.
>>> transformed_sf = transformer.fit_transform(sf)
```

```
Columns:
text    list

Rows: 1

Data:
+-----+
|      text      |
+-----+
| [This is sentence 1., This... |
+-----+
[1 rows x 1 columns]
```

Image features

These feature transformations are useful when you have image data.

- [Deep Feature Extractor](#)

Deep Feature Extractor

Takes an input dataset, propagates each example through the network, and returns an SArray of dense feature vectors. These feature vectors can be used as input to train another classifier such as a LogisticClassifier, SVMClassifier, BoostedTreesClassifier, or NeuralNetClassifier.

Deep features can be used to extract features from your own models or using a pre-trained model for ImageNet (NIPS 2012, Alex Krizhevsky et al.). Turi provides a free pre-trained model for use as demonstrated below.

Introductory Example

```
# Create data.
import graphlab as gl

# Import data from MNIST
data = gl.SFrame('https://static.turi.com/datasets/mnist/sframe/train6k')

# Create a DeepFeatureExtractorObject
#If `model='auto'` is used, an appropriate model is chosen from a collection
#of pre-trained models hosted by Turi.
extractor = gl.feature_engineering.DeepFeatureExtractor(features = 'image',
                                                       model='auto')

# Fit the encoder for a given dataset.
extractor = extractor.fit(data)

# Return the model used for the deep feature extraction.
extracted_model = extractor['model']
```

Once a DeepFeatureExtractor object is constructed, it must first be fitted and then the transform function can be called to extract features. The extracted features can then be used as a part of a LogisticClassifier.

```
# Extract features.
features_sf = extractor.transform(data)

+-----+-----+-----+
| label |      image      | deep_features_image |
+-----+-----+-----+
|   5   | Height: 28 Width: 28 | [0.0531935989857, 0.653152... |
|   8   | Height: 28 Width: 28 | [0.0531935989857, 1.006503... |
|   1   | Height: 28 Width: 28 | [0.0531935989857, 0.053193... |
|   4   | Height: 28 Width: 28 | [0.0531935989857, 0.063806... |
|   2   | Height: 28 Width: 28 | [0.0531935989857, 0.347246... |
|   7   | Height: 28 Width: 28 | [0.0531935989857, 0.758747... |
|   0   | Height: 28 Width: 28 | [0.0531935989857, 0.252766... |
|   2   | Height: 28 Width: 28 | [0.0531935989857, 0.526395... |
|   5   | Height: 28 Width: 28 | [0.0531935989857, 0.053193... |
|   9   | Height: 28 Width: 28 | [0.0531935989857, 1.276176... |
+-----+-----+-----+
[6000 rows x 3 columns]

# Train a classifier using the deep features!.
model = gl.logistic_classifier.create(features_sf, target='label',
                                         features = ['deep_features_image'])
```

Other transformations

These are useful when doing feature engineering.

- [Hasher](#)
- [Random Projection](#)
- [Transformer Chain](#)
- [Custom Transformer](#)

Feature Hashing

Hashes an input feature space to an n-bit feature space. Feature hashing is an efficient way of vectorizing features, and performing dimensionality reduction or expansion along the way. Supported types include `array.array`, `list`, `dict`, `float`, `int`, and `string`. The behaviour for different input data column types is as follows:

- **array.array** : The index of each element is combined with the column name and hashed, and the element becomes the value.
- **list** : Behaves the same as array.array; if the element is non-numerical, the element is combined with the column name and hashed, and 1 is used as the value.
- **dict** : Each key in the dictionary is combined with the column name and hashed, and the value is kept. If the value is non-numerical, the element is combined with the column name and hashed, and 1 is used as the value.
- **float** : The column name is hashed, and the column entry becomes the value.
- **int** : Same behavior as float.
- **string** : Hash the string and use it as a key, and use 1 as the value.

The hashed values are collapsed into a single sparse representation of a vector. The `num_bits` parameter specifies the number of bits to hash to.

Note: Each time an entry is hashed, a separate hash on the key is performed to either add or subtract a value with equal probability. This keeps the value unbiased, since the expectation value for each feature (across all examples) is 0.

Introductory Example

```

from graphlab.toolkits.feature_engineering import *

# Construct a feature hasher with default options.
sf = graphlab.SFrame({'a': [1,2,3], 'b' : [2,3,4], 'c': [9,10,11]})
hasher = graphlab.feature_engineering.create(sf, FeatureHasher())

# Transform the data using the hasher.
hashed_sf = hasher.transform(sf)

# Save the transformer.
hasher.save('save-path')

# Hash only a single column 'a'.
hasher = graphlab.feature_engineering.create(sf,
                                              FeatureHasher(features = ['a']))

# Hash all columns except 'a'.
hasher = graphlab.feature_engineering.create(sf,
                                              FeatureHasher(excluded_features = ['a']))

```

Fitting and transforming

Once a FeatureHasher object is constructed, it must first be fitted and then the transform function can be called to generate hashed features.

For numeric columns:

```

sf = graphlab.SFrame({'a' : [1,2,3], 'b' : [2,3,4]})
hasher = graphlab.feature_engineering.FeatureHasher()
fit_hasher = hasher.fit(sf)
hashed_sf = fit_hasher.transform(sf)

```

Columns:
 hashed_features dict

Rows: 3

Data:

hashed_features
{79785: -1, 188475: -2}
{79785: -2, 188475: -3}
{79785: -3, 188475: -4}

[3 rows x 1 columns]

For list/vector columns:

```
l1 = [1, 2, 3]
l2 = [2, 3, 4]
sf = graphlab.SFrame({'a' : [l1,l1,l1], 'b' : [l2,l2,l2]})  
hasher = graphlab.feature_engineering.FeatureHasher()  
fit_hasher = hasher.fit(sf)  
hashed_sf = fit_hasher.transform(sf)
```

Columns:
 hashed_features dict

Rows: 3

Data:

hashed_features
{642: 2.0, 164: -3.0, 937:...}
{642: 2.0, 164: -3.0, 937:...}
{642: 2.0, 164: -3.0, 937:...}

[3 rows x 1 columns]

For string columns:

```
sf = graphlab.SFrame({'a' : ['a','b','c'], 'b' : ['d','e','f']})  
hasher = graphlab.feature_engineering.FeatureHasher()  
fit_hasher = hasher.fit(sf)  
hashed_sf = fit_hasher.transform(sf)
```

Columns:
 hashed_features dict

Rows: 3

Data:

hashed_features
{405: 1, 79: 1}
{454: 1, 423: 1}
{308: 1, 36: 1}

[3 rows x 1 columns]

For dictionary columns:

```
dict1 = {'a' : 1 , 'b' : 2 , 'c' : 3}
dict2 = {'d' : 4 , 'e' : 5 , 'f' : 6}
sf = graphlab.SFrame({'a' : [dict1, dict1, dict1],
                      'b' : [dict2, dict2, dict2]})

hasher = graphlab.feature_engineering.FeatureHasher()
fit_hasher = hasher.fit(sf)
hashed_sf = fit_hasher.transform(sf)
```

Columns:
 hashed_features dict

Rows: 3

Data:

hashed_features
{36: 3, 454: 5, 423: 2, 79...}
{36: 3, 454: 5, 423: 2, 79...}
{36: 3, 454: 5, 423: 2, 79...}

[3 rows x 1 columns]

Random Projection

Random projection is a tool for representing high-dimensional data in a low-dimensional feature space, typically for data visualization or methods that rely on fast computation of pairwise distances, like nearest neighbors searching and nonparametric clustering.

Introduction

Dimension reduction is a very useful preprocessing step for several tasks:

1. **Data visualization.** For data with more than three features (four or five if we're really clever, e.g. [Gapminder](#)), it's impossible to plot in the original feature space. Dimension reduction gives us a two-dimension representation of the data that can be easily plotted.
2. **Analyzing high-dimensional data.** The curse of dimensionality makes it difficult to tell when high-dimensional data points are similar to each other or not. This is a particularly acute problem for nearest neighbors search and nonparametric clustering, where random projection is often a standard preprocessing step.
3. **Storing and moving large datasets.** Dimension reduction allows us to reduce the size of datasets without discarding any data points, as we would have to do with other techniques like vector quantization or locality-sensitive hashing.

GraphLab Create's [RandomProjection](#) tool uses **Gaussian random projection**. Let's say we have a numeric dataset with n examples, each of which is represented by d features (where d is presumably relatively large, maybe on the order of hundreds or thousands). In other words, our data is a matrix X , with n rows and d columns. Suppose we want to reduce the dimensionality of our data so that each example is represented by only k features, where k is small, like 2 or 10.

For Gaussian random projection we construct a projection matrix R with d rows and k columns. Each entry is independently sampled from a standard Gaussian distribution

$$R_{ij} \sim N(0, 1)$$

The projection is done by multiplying our data matrix by the projection matrix:

$$Y = \frac{1}{\sqrt{k}} X R$$

so that our output dataset Y has n rows with only k columns. The scalar $1/\sqrt{k}$ ensures that the Euclidean distance between any two points in the new low-dimensional space is very close to the distance between the same points in the original high-dimensional space, with high probability.

Usage

To illustrate usage of the similarity search toolkit, we use a small subset of the [MNIST handwritten digits image dataset](#), which can be downloaded from the public Turi datasets bucket on Amazon S3. The download is about 1.5 MB.

```
import graphlab as gl
import os

if os.path.exists('mnist_train6k'):
    mnist = gl.SFrame('mnist_train6k')
else:
    mnist = gl.SFrame('https://static.turi.com/datasets/mnist/sframe/train6k')
    mnist.save('mnist_train6k')
```

The first preprocessing step is to convert the image data to an array of floats. Each array has 784 entries.

```
import array
mnist['array'] = mnist['image'].astype(array.array)
mnist.print_rows(5)

print "Number of features:", len(mnist['array'][0])
```

label	image	array
5	Height: 28 Width: 28	[0.0, 0.0, 0.0, 0.0, 0.0, ...]
8	Height: 28 Width: 28	[0.0, 0.0, 0.0, 0.0, 0.0, ...]
1	Height: 28 Width: 28	[0.0, 0.0, 0.0, 0.0, 0.0, ...]
4	Height: 28 Width: 28	[0.0, 0.0, 0.0, 0.0, 0.0, ...]
2	Height: 28 Width: 28	[0.0, 0.0, 0.0, 0.0, 0.0, ...]

[6000 rows x 3 columns]

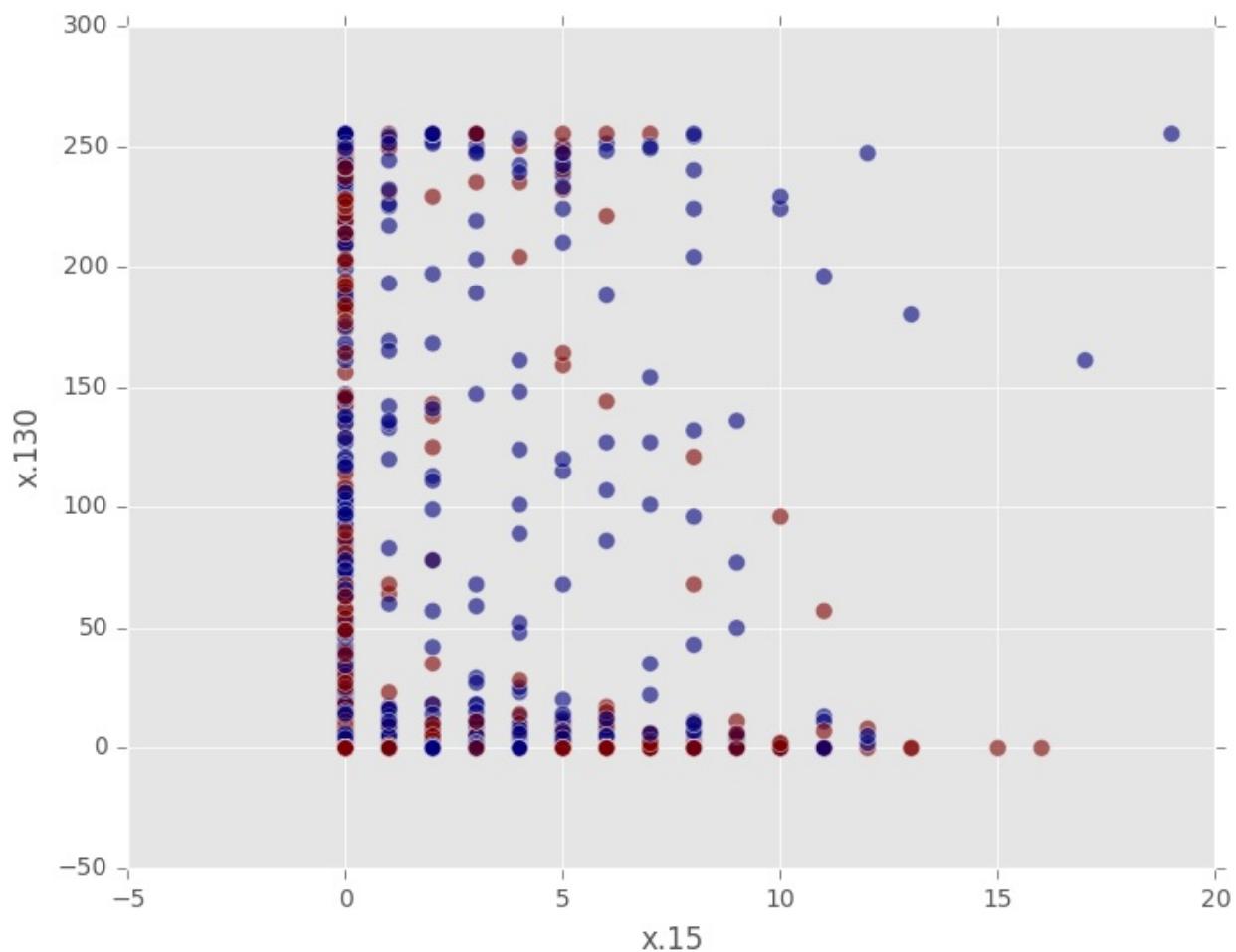
Number of features: 784

With 784 features, this dataset is impossible to plot. A naive thing to try is to pick two features arbitrarily to plot, but even when we limit the plot to just two of the 10 classes, the result is useless. We cannot distinguish the distributions visually.

```
import matplotlib.pyplot as plt
plt.style.use('ggplot')

mnist_temp = mnist.unpack('array', column_name_prefix='x')
mnist_temp = mnist_temp.filter_by([0, 1], 'label')

fig, ax = plt.subplots()
ax.scatter(mnist_temp['x.15'], mnist_temp['x.130'], c=mnist_temp['label'],
           s=50, alpha=0.6)
ax.set_xlabel('x.15'); ax.set_ylabel('x.130')
fig.show()
```



On the other hand, if we use a Gaussian random projection of the data into 2-dimensional space, we can see the classes much more clearly. Note that we do not need to unpack the input data manually; the `RandomProjection` transformer takes care of that for us. The data returned by the `RandomProjection` transformer is also packed into a single column of array type. To access each entry separately, this column needs to be unpacked.

```

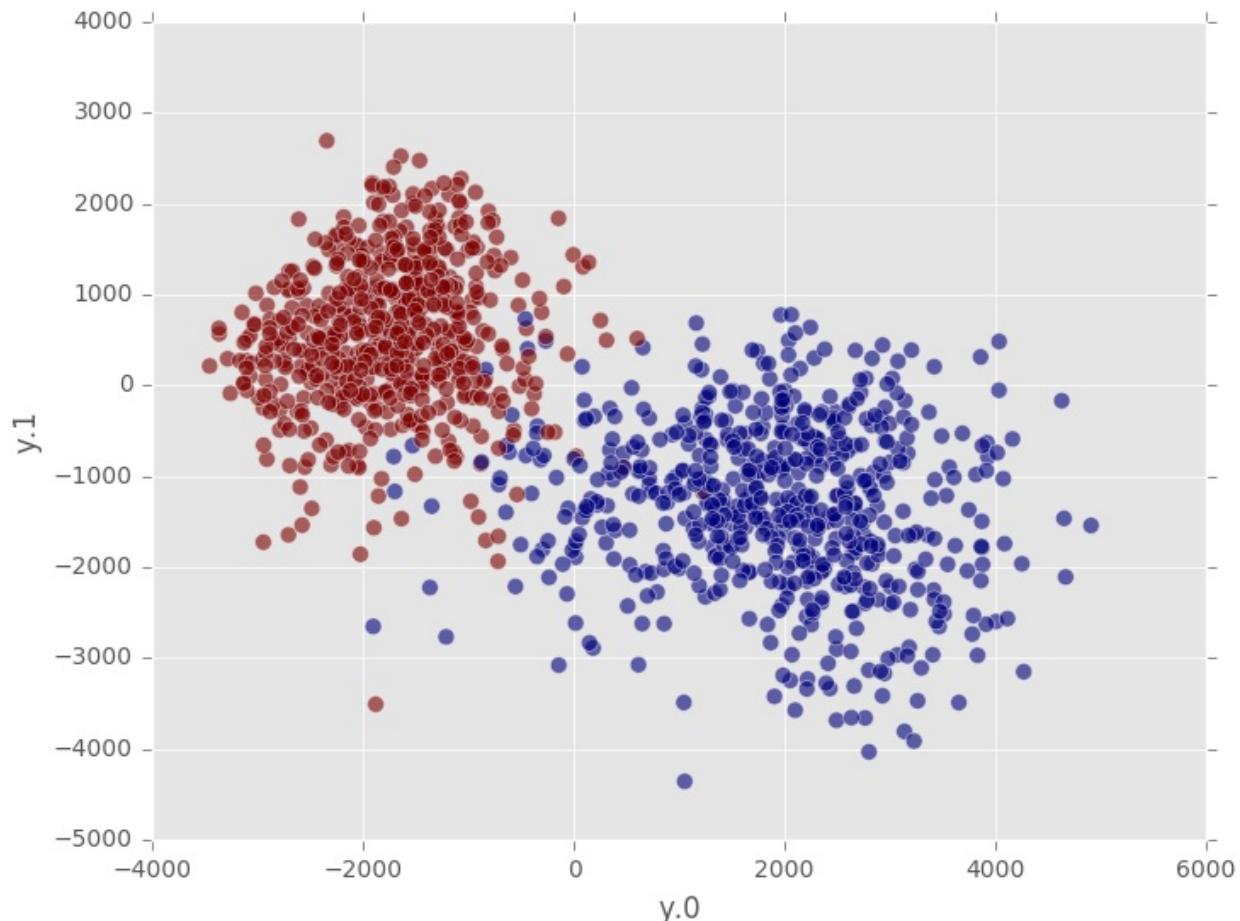
from graphlab.toolkits.feature_engineering import RandomProjection

rp = RandomProjection(features=['array'], embedding_dimension=2,
                      random_seed=19)

mnist_embedded = rp.fit_transform(mnist)
mnist_embedded = mnist_embedded.unpack('embedded_features',
                                       column_name_prefix='y')
mnist_embedded = mnist_embedded.filter_by([0, 1], 'label')

fig, ax = plt.subplots()
ax.scatter(mnist_embedded['y.0'], mnist_embedded['y.1'], c=mnist_embedded['label'],
           s=50, alpha=0.6)
ax.set_xlabel('y.0'); ax.set_ylabel('y.1')
fig.show()

```



The `summary` method of a fitted `RandomProjection` object is useful for understanding how many features the transformer expects to receive for calls to `transform`, as well as the random seed for construction of the projection matrix. Two `RandomProjection` instances with the same random seed yield the same output point for the same input data.

```
rp.summary()
```

```
Class : RandomProjection

Model fields
-----
Embedding dimension : 2
Original dimension : 784
Features : ['array']
Excluded features : None
Output column name : embedded_features
Random seed : 19
Has been fitted : 1
```

References

- Achlioptas, D. (2003). [Database-friendly random projections: Johnson-Lindenstrauss with binary coins (<https://users.soe.ucsc.edu/~optas/papers/jl.pdf>)]. *Journal of Computer and System Sciences*, 66(4).
- Li, P., Hastie, T. J., & Church, K. W. (2006). [Very sparse random projections](#). Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '06.
- Wikipedia. [Dimensionality reduction](#).
- Wikipedia. [Random projection](#).

Transformer Chain

Sequentially apply a list of transforms. Each of the individual steps in the chain must be transformers (ie., a child class of TransformerBase), which can be one of the following:

- Native transformer modules in GraphLab Create (e.g. FeatureHasher).
- User-created modules (defined by inheriting TransformerBase).

Introductory Example

```
# Create data.
sf = graphlab.SFrame({'a': [1,2,3], 'b' : [2,3,4]})

# Create a chain a transformers.
from graphlab.toolkits.feature_engineering import *

# Create a chain of transformers.
chain = graphlab.feature_engineering.create(sf,[QuadraticFeatures(),
                                                FeatureHasher() ])

# Create a chain of transformers with names for each of the steps.
chain = graphlab.feature_engineering.create(sf,[('quadratic', QuadraticFeatures()),
                                                ('hasher', FeatureHasher())])

# Transform the data.
transformed_sf = chain.transform(sf)

# Save the transformer.
chain.save('save-path')

# Access each of the steps in the transformer by name or index
steps = chain['steps']
steps = chain['steps_by_name']
```

Custom Transformer

You can define Transformers in addition to the ones provided by GraphLab Create by inheriting from `TransformerBase`. You must implement the `init`, `fit`, and `transform` methods.

Introductory Example

```
import graphlab
from graphlab.toolkits.feature_engineering import TransformerBase

class MyTransformer(TransformerBase):

    def __init__(self):
        pass

    def fit(self, dataset):
        ''' Learn means during the fit stage.'''
        self.mean = {}
        for col in dataset.column_names():
            self.mean[col] = dataset[col].mean()
        return self

    def transform(self, dataset):
        ''' Subtract means during the transform stage.'''
        for col in dataset.column_names():
            dataset[col] = dataset[col] - dataset[col].mean()
        return dataset

# Dataset
dataset = graphlab.SFrame({"a": range(100)})

# Create the model
model = graphlab.feature_engineering.create(dataset, MyTransformer())

# Transform new data
transformed_sf = model.transform(dataset)

# Save and load this model.
model.save('foo-bar')
```

Modeling Data

Models of your data can provide a principled way of making predictions from the data you have observed. GraphLab Create offers toolkits for easily creating models for common tasks, such as:

- predicting numeric quantities
- making recommendation systems
- clustering data and documents
- analyzing graphs

Classification

Classification is the problem of predicting a **categorical target** using training data. The key difference between **regression** and **classification** is that in regression the target is continuous while in classification, the target is categorical.

Creating classification models is easy with GraphLab Create! Currently, the following models are supported for classification:

- [Logistic regression](#)
- [Nearest neighbor classifier](#)
- [Support vector machines \(SVM\)](#)
- [Boosted Decision Trees](#)
- [Random Forests](#)
- [Decision Tree](#)
- [Neural network classifier \(deep learning\)](#)

These algorithms differ in how they make predictions, but conform to the same API. With all models, call **create()** to create a model, **predict()** to make flexible predictions on the returned model, **classify()** which provides all the sufficient statistics for classifying data, and **evaluate()** to measure performance of the predictions. Models can incorporate:

- Numeric features
- Categorical variables
- Dictionary features (i.e sparse features)
- List features (i.e dense arrays)
- Text data
- Images

Model Selector

It isn't always clear that we know exactly which model is suitable for a given task. GraphLab Create's model selector automatically picks the right model for you based on statistics collected from the data set.

```
import graphlab as gl

# Load the data
data = gl.SFrame('https://static.turi.com/datasets/regression/yelp-data.csv')

# Restaurants with rating >=3 are good
data['is_good'] = data['stars'] >= 3

# Make a train-test split
train_data, test_data = data.random_split(0.8)

# Automatically picks the right model based on your data.
model = gl.classifier.create(train_data, target='is_good',
                             features = ['user_avg_stars',
                                         'business_avg_stars',
                                         'user_review_count',
                                         'business_review_count'])

# Generate predictions (class/probabilities etc.), contained in an SFrame.
predictions = model.classify(test_data)

# Evaluate the model, with the results stored in a dictionary
results = model.evaluate(test_data)
```

GraphLab Create implementations are built to work with up to billions of examples and up to millions of features.

#Logistic Regression Logistic regression is a regression model that is popularly used for classification tasks. In logistic regression, the probability that a **binary target is True** is modeled as a [logistic function](http://en.wikipedia.org/wiki/Logistic_function) of a linear combination of features. The following figure illustrates how logistic regression is used to train a 1-dimensional classifier. The training data consists of positive examples (depicted in blue) and negative examples (in orange). The decision boundary (depicted in pink) separates out the data into two classes.

Background

Given a set of features x_i , and a label $y_i \in \{0, 1\}$, logistic regression interprets the probability that the label is in one class as a logistic function of a linear combination of the features:

$$f_i(\theta) = p(y_i = 1|x) = \frac{1}{1 + \exp(-\theta^T x)}$$

Analogous to linear regression, an intercept term is added by appending a column of 1's to the features and L1 and L2 regularizers are supported. The composite objective being optimized for is the following:

$$\min_{\theta} \sum_{i=1}^n f_i(\theta) + \lambda_1 \|\theta\|_1 + \lambda_2 \|\theta\|_2^2$$

where λ_1 is the `L1_penalty` and λ_2 is the `L2_penalty`.

Introductory Example

First, let's construct a binary target variable. In this example, we will predict **if a restaurant is good or bad**, with 1 and 2 star ratings indicating a bad business and 3-5 star ratings indicating a good one. We will use the following features.

- Average rating of a given business
- Average rating made by a user
- Number of reviews made by a user
- Number of reviews that concern a business

```

import graphlab as gl

# Load the data
data = gl.SFrame('https://static.turi.com/datasets/regression/yelp-data.csv')

# Restaurants with rating >=3 are good
data['is_good'] = data['stars'] >= 3

# Make a train-test split
train_data, test_data = data.random_split(0.8)

# Create a model.
model = gl.logistic_classifier.create(train_data, target='is_good',
                                       features = ['user_avg_stars',
                                                    'business_avg_stars',
                                                    'user_review_count',
                                                    'business_review_count'])

# Save predictions (probability estimates) to an SArray
predictions = model.classify(test_data)

# Evaluate the model and save the results into a dictionary
results = model.evaluate(test_data)

```

Advanced Usage

Refer to the chapter on linear regression for the following features:

- Accessing attributes of the model
- Interpreting results
- Using categorical features
- Sparse features
- List features
- Feature rescaling
- Chosing the solver
- Regularizing models

We will now discuss some advanced features that are **specific to logistic regression**.

Making Predictions

Predictions using a GraphLab Create classifier is easy. The **classify()** method provides a one-stop shop for all that you need from a classifier.

- A class prediction
- Probability/Confidence associated with that class prediction.

In the following example, the first prediction was class **1** with a **90.5%** probability.

```
predictions = model.classify(test_data)
print predictions
```

```
+-----+
| class | probability |
+-----+
| 1 | 0.905618772131 |
| 1 | 0.941576249302 |
| 1 | 0.948254387657 |
| 0 | 0.996952633956 |
| 1 | 0.944229260472 |
| 1 | 0.951769966846 |
| 1 | 0.905561314917 |
| 1 | 0.957697429248 |
| 1 | 0.98527411871 |
| 1 | 0.973282185166 |
| ... | ... |
+-----+
[43018 rows x 2 columns]
```

Note: Only the head of the SFrame is printed.
 You can use `print_rows(num_rows=m, num_columns=n)` to print more rows and columns.

Making detailed predictions

Logistic regression predictions can take one of three forms:

- **Classes** (default): Thresholds the probability estimate at 0.5 to predict a class label i.e. **0/1**.
- **Probabilities**: A probability estimate (in the range [0,1]) that the example is in the **True** class. Note that this is not the same as the probability estimate in the `classify` function.
- **Margins** : Distance to the linear decision boundary learned by the model. The larger the distance, the more confidence we have that it belongs to one class or the other.

GraphLab Create's logistic regression model can return `predictions` for any of these types:

```
pred_class = model.predict(test_data, output_type = "class")           # Class
pred_prob_one = model.predict(test_data, output_type = 'probability') # Probability
pred_margin = model.predict(test_data, output_type = "margin")         # Margins
```

Evaluating Results

We can also evaluate our predictions by comparing them to known ratings. The results are evaluated using two metrics:

- **Classification Accuracy**: Fraction of test set examples with correct class label predictions.

- **Confusion Matrix:** Cross-tabulation of predicted and actual class labels.

```
result = model.evaluate(test_data)
print "Accuracy      : %s" % result['accuracy']
print "Confusion Matrix : \n%s" % result['confusion_matrix']
```

```
Accuracy      : 0.860862092991
Confusion Matrix :
+-----+-----+-----+
| target_label | predicted_label | count |
+-----+-----+-----+
|      0       |        0       |  2348 |
|      0       |        1       |  4816 |
|      1       |        0       |   912 |
|      1       |        1       | 34942 |
+-----+-----+-----+
[4 rows x 3 columns]
```

Working with imbalanced data

Many difficult **real-world** problems have imbalanced data, where at least one class is under-represented. GraphLab Create models can handle the imbalanced data by assigning asymmetric costs of misclassifying elements of different classes.

```
# The data can be downloaded using
data = gl.SFrame.read_csv('https://static.turi.com/datasets/xgboost/mushroom.csv')

# Label 'c' is edible
data['label'] = data['label'] == 'c'

# Make a train-test split
train_data, test_data = data.random_split(0.8)

# Create a model which weights classes based on frequency in the training data.
model = gl.logistic_classifier.create(train_data, target='label',
                                       class_weights = 'auto')
```

Multiclass Classification

Multiclass classification is the problem of classifying instances into one of many (i.e more than two) possible instances. As an example, binary classification can be used to train a classifier that can distinguish between two classes, say "cat" or "dog", while multiclass classification can be used to train a finite set of labels at the same time, say "cat", "dog", "rat", and "cow".

```
import graphlab as gl

# Load the data
# The data can be downloaded using
data = gl.SFrame('https://static.turi.com/datasets/mnist/sframe/train6k-array')

# Make a train-test split
train_data, test_data = data.random_split(0.8)

# Create a model.
model = gl.logistic_classifier.create(train_data, target='label')

# Save predictions to an SFrame (class and corresponding class-probabilities)
predictions = model.classify(test_data)

# Top 5 predictions with probabilities, rank, and margin
top = model.predict_topk(test_data, output_type='probability', k = 5)
top = model.predict_topk(test_data, output_type='rank', k = 5)
top = model.predict_topk(test_data, output_type='margin', k = 5)

# Evaluate the model and save the results into a dictionary
results = model.evaluate(test_data)
```

Top-k predictions

Multiclass classification provides the top-k class predictions for each class. The predictions are either margins, probabilities, or a rank for the predicted class for each example. In the following example, we provide the top 5 predictions, ordered by class probability, for each data point in the test set.

```
top = model.predict_topk(test_data, output_type='probability', k = 3)
print top
```

```
Columns:
 id    str
 class  str
 probability float

Rows: 3711

Data:
+-----+-----+
| id | class | probability |
+-----+-----+
| 0  | 5    | 0.94296406887 |
| 0  | 1    | 0.0140330526641 |
| 0  | 8    | 0.00636249767982 |
| 1  | 8    | 0.929146865934 |
| 1  | 9    | 0.0139581314344 |
| 1  | 1    | 0.00982837828507 |
| 2  | 1    | 0.937192457289 |
| 2  | 7    | 0.0106293228679 |
| 2  | 4    | 0.00910849289074 |
| 3  | 5    | 0.900146607924 |
| ... | ... | ...
+-----+
```

Nearest Neighbor Classifier

The [nearest neighbor classifier](#) is one of the simplest classification models, but it often performs nearly as well as more sophisticated methods.

Background

The nearest neighbors classifier predicts the class of a data point to be the most common class among that point's neighbors. Suppose we have n training data points, where the i 'th point has both a vector of features x_i and class label y_i . For a new point x^* , the nearest neighbor classifier first finds the set of neighbors of x^* , denoted $N(x^*)$. The class label for x^* is then predicted to be

$$y^* = \max_c \sum_{i \in N(x^*)} I(y_i = c)$$

where the indicator function $I()$ is 1 if the argument is true, and 0 otherwise. The simplicity of this approach makes the model relatively straightforward to understand and communicate to others, and naturally lends itself to multi-class classification.

Defining the criteria for the **neighborhood** of a prediction data point requires careful thought and domain knowledge. A function must be specified to measure the distance between any two data points, and then the size of "neighborhoods" relative to this distance function must be set.

For the first step, there are many standard distance functions (e.g. Euclidean, Jaccard, Levenshtein) that work well for data whose features are all of the same type, but for heterogeneous data the task is a bit trickier. GraphLab Create overcomes this problem with **composite distances**, which are weighted sums of standard distance functions applied to appropriate subsets of features. For more about distance functions in GraphLab Create, including composite distances, please see the [API documentation for the distances module](#). The end of this chapter describes how to use a composite distance with the nearest neighbor classifier in particular.

Once the distance function is defined, the user must indicate the criteria for deciding when training data are in the "neighborhood" of a prediction point. This is done by setting two constraints:

1. `radius` - the maximum distance a training example can be from the prediction point and still be considered a neighbor, and

2. `max_neighbors` - the maximum number of neighbors for the prediction point. If there are more points within `radius` of the prediction point, the closest `max_neighbors` are used.

Unlike the other classifiers in the GraphLab Create classifier toolkit, the nearest neighbors classifiers is an **instance-based** method, which means that the model must store all of the training data. For each prediction, the model must search all of the training data to find the neighbor points in the training data. GraphLab Create performs this search intelligently, but predictions are nevertheless typically slower than other classification models.

Basic Example

To illustrate basic usage of the nearest neighbor classifier, we again use the Yelp restaurant review data, with the goal of predicting how many "stars" a user will give a particular business. Anticipating that we will want to test the validity of the model, we first split the data into training and testing subsets.

```
import graphlab as gl
import os

filename = 'yelp-data.csv'

if os.path.exists(filename):
    data = gl.SFrame.read_csv(filename)
else:
    data = gl.SFrame('https://static.turi.com/datasets/regression/{}'.format(filename))
    data.save(filename, format='csv')

train_data, test_data = data.random_split(0.9)
```

In this example we build a classifier using only the four numeric features listed in the logistic regression chapter, namely the average stars awarded by each user and to each business and the total count of reviews given by each user and to each business. The review counts features are typically much larger than the average stars features, which would cause the review counts to dominate standard numeric distance functions. To avoid this we standardize the features before creating the model.

```

numeric_features = ['user_avg_stars',
                    'business_avg_stars',
                    'user_review_count',
                    'business_review_count']

for ftr in numeric_features:
    mean = train_data[ftr].mean()
    stdev = train_data[ftr].std()
    train_data[ftr] = (train_data[ftr] - mean) / stdev
    test_data[ftr] = (test_data[ftr] - mean) / stdev

```

Finally, we create the model and generate predictions.

```

m = gl.nearest_neighbor_classifier.create(train_data, target='stars',
                                         features=numeric_features)
predictions = m.classify(test_data, max_neighbors=20, radius=None)
print predictions

```

class	probability
4	0.65
4	0.5
5	0.8
5	0.65
5	1.0
4	0.55
4	0.35
3	0.45
4	0.45
5	0.4

[21466 rows x 2 columns]

Note: Only the head of the SFrame is printed.
 You can use `print_rows(num_rows=m, num_columns=n)` to print more rows and columns.

Advanced Usage

The `classify` method returns an SFrame with both the predicted class and the probability score of that class, which is simply the fraction of the points' neighbors which belong to the most common class. As with multiclass logistic regression, the `predict_topk` method can be used to see the fraction of neighbors belonging to every target class.

```

topk = m.predict_topk(test_data[:5], max_neighbors=20, k=3)
print topk

```

```
## -- End pasted text --
+-----+-----+-----+
| row_id | class | probability |
+-----+-----+-----+
| 0      | 5     | 0.7       |
| 0      | 4     | 0.3       |
| 3      | 4     | 0.45      |
| 3      | 5     | 0.3       |
| 3      | 3     | 0.2       |
| 1      | 5     | 0.8       |
| 1      | 4     | 0.15      |
| 1      | 3     | 0.05      |
| 2      | 4     | 0.6       |
| 2      | 5     | 0.25      |
+-----+-----+-----+
[14 rows x 3 columns]
Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.
```

To get a sense of the model validity, pass the test data to the `evaluate` method.

```
evals = m.evaluate(test_data[:3000])
print evals['accuracy']
```

```
0.46
```

46% accuracy seems low, but remember that we are in a multi-class classification setting. The most common class (4 stars) only occurs in 34.8% of the test data, so our model has indeed learned something. The confusion matrix produced by the `evaluate` method can help us to better understand the model performance. In this case we see that 83.9% of our predictions are actually within 1 star of the true number of stars.

```
conf_matrix = evals['confusion_matrix']
conf_matrix['within_one'] = conf_matrix.apply(
    lambda x: abs(x['target_label'] - x['predicted_label']) <= 1)
num_within_one = conf_matrix[conf_matrix['within_one']]['count'].sum()
print float(num_within_one) / len(test_data)
```

```
0.8386693230783487
```

Suppose we want to add the `text` column as a feature. One way to do this is to treat each entry as a "bag of words" by simply counting the number of times each word appears but ignoring the order (please see the text analytics chapter for more detail).

```
train_data['word_counts'] = gl.text_analytics.count_words(train_data['text'],
                                                       to_lower=True)
test_data['word_counts'] = gl.text_analytics.count_words(test_data['text'],
                                                       to_lower=True)
```

For example, the (abbreviated) text of the first review in the training set is:

My wife took me here on my birthday for breakfast and it was excellent. The weather was perfect which made sitting outside overlooking their grounds an absolute pleasure....

while the (also abbreviated) bag-of-words representation is a dictionary that maps each word to the number of times that word appears:

```
{'2': 1, 'a': 1, 'absolute': 1, 'absolutely': 1, 'amazing': 2, 'an': 1,
 'and': 8, 'anyway': 1, 'arrived': 1, 'back': 1, 'best': 2, 'better': 1,
 'birthday': 1, 'blend': 1, 'bloody': 1, 'bread': 1, 'breakfast': 1, ... }
```

The `weighted_jaccard` distance measures the difference between two sets, weighted by the counts of each element (please see the [API documentation](#) for details). To combine this output with the numeric distance we used above, we specify a **composite distance**. Each element in this list includes a list (or tuple) of feature names, a standard distance function name, and a numeric weight. The weight on each component can be adjusted to produce the same effect as normalizing features.

```
my_dist = [
    [numeric_features, 'euclidean', 1.0],
    [['word_counts'], 'weighted_jaccard', 1.0]
]

m2 = gl.nearest_neighbor_classifier.create(train_data, target='stars',
                                           distance=my_dist)
accuracy = m2.evaluate(test_data[:3000], metric='accuracy')
print accuracy
```

```
{'accuracy': 0.482}
```

Adding the text feature appears to slightly improve the accuracy of our classifier. For more information, please see the following resources:

- [User Guide chapter on nearest neighbors search](#)
- [API documentation on the nearest neighbor classifier](#)
- [API documentation on the distances module](#)

- [Wikipedia on the k-nearest neighbors algorithm](#)

Support Vector Machines Support Vector Machines (SVM) is another popular model used for classification tasks. In logistic regression, the probability that a **binary target is True** is modeled as a [logistic function](http://en.wikipedia.org/wiki/Logistic_function) of the features. The following figure illustrates how an SVM is used to train a 2-dimensional classifier. The training data consists of positive examples (depicted in orange) and negative examples (in blue). The decision boundary (depicted in pink) separates out the data into two classes.

Background

Currently, GraphLab Create implements a linear C-SVM (SVC). In this model, given a set of features x_i , and a label $y_i \in \{0, 1\}$ the linear SVM minimizes the loss function:

$$f_i(\theta) = \max(1 - \theta^T x, 0)$$

As with other models, an intercept term is added by appending a column of 1's to the features. The composite objective being optimized for is the following:

$$\min_{\theta} \lambda \sum_{i=1}^n f_i(\theta) + \|\theta\|_2^2$$

where λ is the `penalty` parameter (the C in the C-SVM) that determines the weight in the loss function towards the regularizer. The larger the value of λ , the more is the weight given to the mis-classification loss. GraphLab Create solves the Linear-SVM formulation by approximating the hinge-loss with a smooth function (see [Zhang et. al.](#) for details).

Introductory Example

Using the same example as we did for logistic regression, we will predict **if a restaurant is good or bad**, with 1 and 2 star ratings indicating a bad business and 3-5 star ratings indicating a good one. We will use the following features:

- Average rating of a given business
- Average rating made by a user
- Number of reviews made by a user
- Number of reviews that concern a business

The usage is similar to the logistic regression module:

```

import graphlab as gl

# Load the data
# The data can be downloaded using
data = gl.SFrame('https://static.turi.com/datasets/regression/yelp-data.csv')

# Restaurants with rating >=3 are good
data['is_good'] = data['stars'] >= 3

# Make a train-test split
train_data, test_data = data.random_split(0.8)

# Create a model.
model = gl.svm_classifier.create(train_data, target='is_good',
                                  features = ['user_avg_stars',
                                              'business_avg_stars',
                                              'user_review_count',
                                              'business_review_count'])

# Save predictions (class only) to an SFrame
predictions = model.predict(test_data)

# Evaluate the model and save the results into a dictionary
results = model.evaluate(test_data)

```

Advanced Usage

Refer to the chapter on linear regression for the following features:

- [Accessing attributes of the model](#)
- [Interpreting results](#)
- [Using categorical features](#)
- [Sparse features](#)
- [List features](#)
- [Feature rescaling](#)
- [Choosing the solver](#)
- [Regularizing models](#)
- [Evaluating Results](#)
- [Working with imbalanced data](#)

We will now discuss some advanced features that are **specific to SVM**.

Making Predictions

Predictions using a GraphLab classifier is easy. The **classify()** method provides a one-stop shop for all that you need from a classifier. In the following example, the first prediction was class 1. Currently, the SVM classifier is not calibrated for probability predictions. Stay tuned

for that feature in an upcoming release.

```
predictions = model.classify(test_data)
print predictions
```

```
+-----+
| class |
+-----+
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| ... |
+-----+
[43414 rows x 1 columns]
Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.
```

Making detailed predictions

SVM predictions can take one of two forms:

- **Margins** : Distance to the linear decision boundary learned by the model. The larger the distance, the more confidence we have that it belongs to one class or the other.
- **Classes** (default) : Thresholds the margin at 0.0 to predict a class label i.e. **0/1**.

SVM does not currently support predictions as probability estimates.

```
pred_class = model.predict(test_data, output_type = "class")      # Class
pred_margin = model.predict(test_data, output_type = "margin")    # Margins
```

Penalty Term

The SVM model contains a `penalty` term on the mis-classification loss of the model. The smaller this weight, the lower is the emphasis placed on misclassified examples which in-turn results in smaller coefficients. The `penalty` term can be set as follows:

```
model = gl.svm_classifier.create(train_data, target='is_good', penalty=100,
                                  features = ['user_avg_stars',
                                              'business_avg_stars',
                                              'user_review_count',
                                              'business_review_count'])
```

Decision Tree Classifier

A decision tree classifier is a simple machine learning model suitable for getting started with classification tasks. Refer to the chapter on [decision tree regression](#) for background on decision trees.

Introductory Example

```
import graphlab as gl

# Load the data
# The data can be downloaded using
data = gl.SFrame.read_csv('https://static.turi.com/datasets/xgboost/mushroom.csv')

# Label 'c' is edible
data['label'] = data['label'] == 'c'

# Make a train-test split
train_data, test_data = data.random_split(0.8)

# Create a model.
model = gl.decision_tree_classifier.create(train_data, target='label',
                                             max_depth = 3)

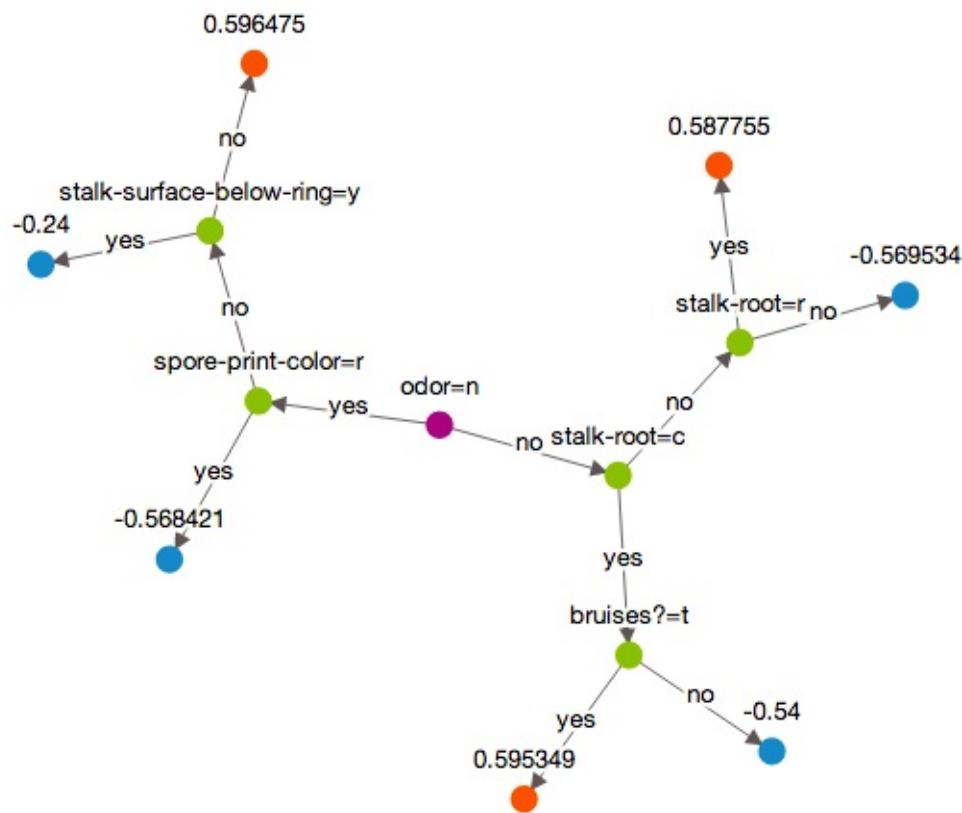
# Save predictions to an SArray.
predictions = model.predict(test_data)

# Evaluate the model and save the results into a dictionary
results = model.evaluate(test_data)
```

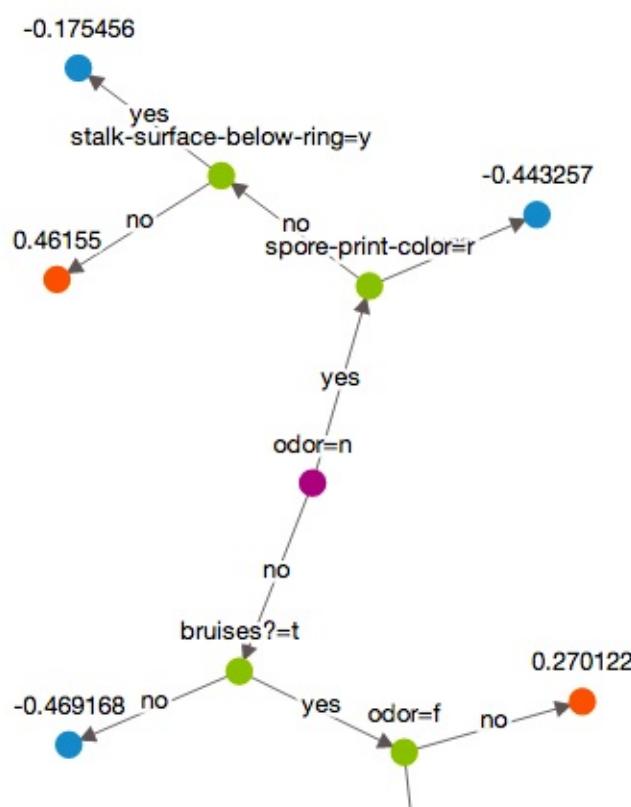
We can visualize the models using

```
model.show(view="Tree", tree_id=0)
model.show(view="Tree", tree_id=1)
```

tree_0



tree_1





Advanced Features

Refer to the earlier chapters for the following features:

- Using categorical features
- Sparse features
- List features
- Evaluating Results
- Multiclass Classification
- Working with imbalanced data

Random Forest Classifier

A Random Forest classifier is one of the most effective machine learning models for predictive analytics. Refer to the chapter on [random forest regression](#) for background on random forests.

Introductory Example

```
import graphlab as gl

# Load the data
# The data can be downloaded using
data = gl.SFrame.read_csv('https://static.turi.com/datasets/xgboost/mushroom.csv')

# Label 'c' is edible
data['label'] = data['label'] == 'c'

# Make a train-test split
train_data, test_data = data.random_split(0.8)

# Create a model.
model = gl.random_forest_classifier.create(train_data, target='label',
                                             max_iterations=2,
                                             max_depth = 3)

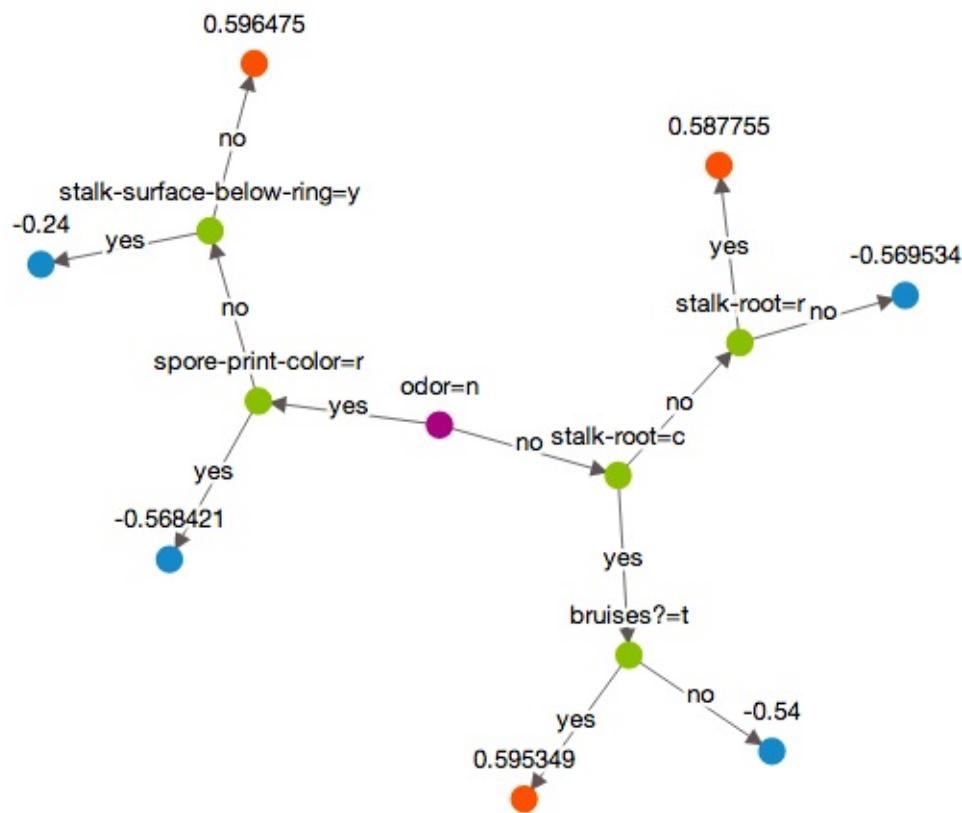
# Save predictions to an SArray.
predictions = model.predict(test_data)

# Evaluate the model and save the results into a dictionary
results = model.evaluate(test_data)
```

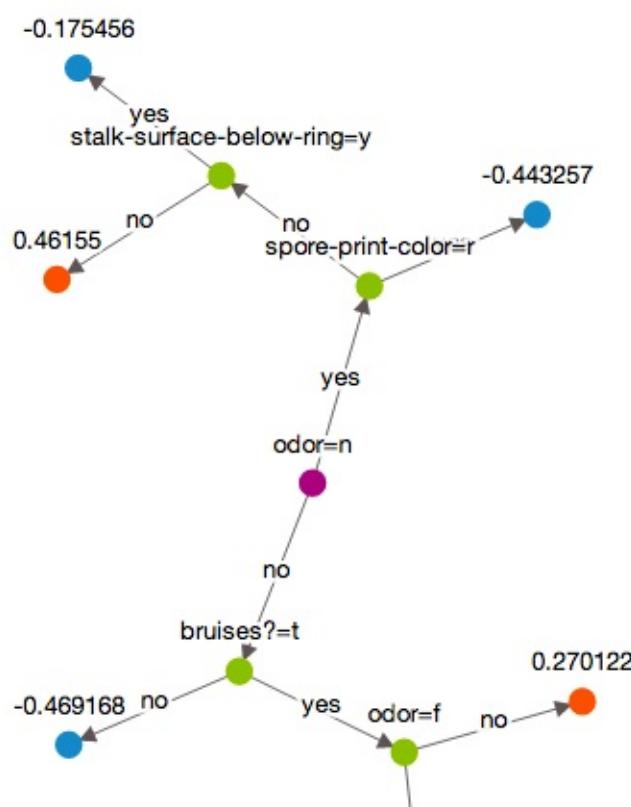
We can visualize the models using

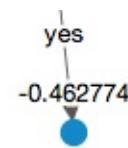
```
model.show(view="Tree", tree_id=0)
model.show(view="Tree", tree_id=1)
```

tree_0



tree_1





See the chapter on [random forest regression](#) for additional tips and tricks of using the random forest classifier model.

Advanced Features

Refer to the earlier chapters for the following features:

- [Using categorical features](#)
- [Sparse features](#)
- [List features](#)
- [Evaluating Results](#)
- [Multiclass Classification](#)
- [Working with imbalanced data](#)

Gradient Boosted Regression Trees

The Gradient Boosted Regression Trees (GBRT) model (also called Gradient Boosted Machine or GBM), is one of the most effective machine learning models for predictive analytics, making it the industrial workhorse for machine learning. Refer to the chapter on [boosted tree regression](#) for background on boosted decision trees.

Introductory Example

```
import graphlab as gl

# Load the data
# The data can be downloaded using
data = gl.SFrame.read_csv('https://static.turi.com/datasets/xgboost/mushroom.csv')

# Label 'c' is edible
data['label'] = data['label'] == 'c'

# Make a train-test split
train_data, test_data = data.random_split(0.8)

# Create a model.
model = gl.boosted_trees_classifier.create(train_data, target='label',
                                             max_iterations=2,
                                             max_depth = 3)

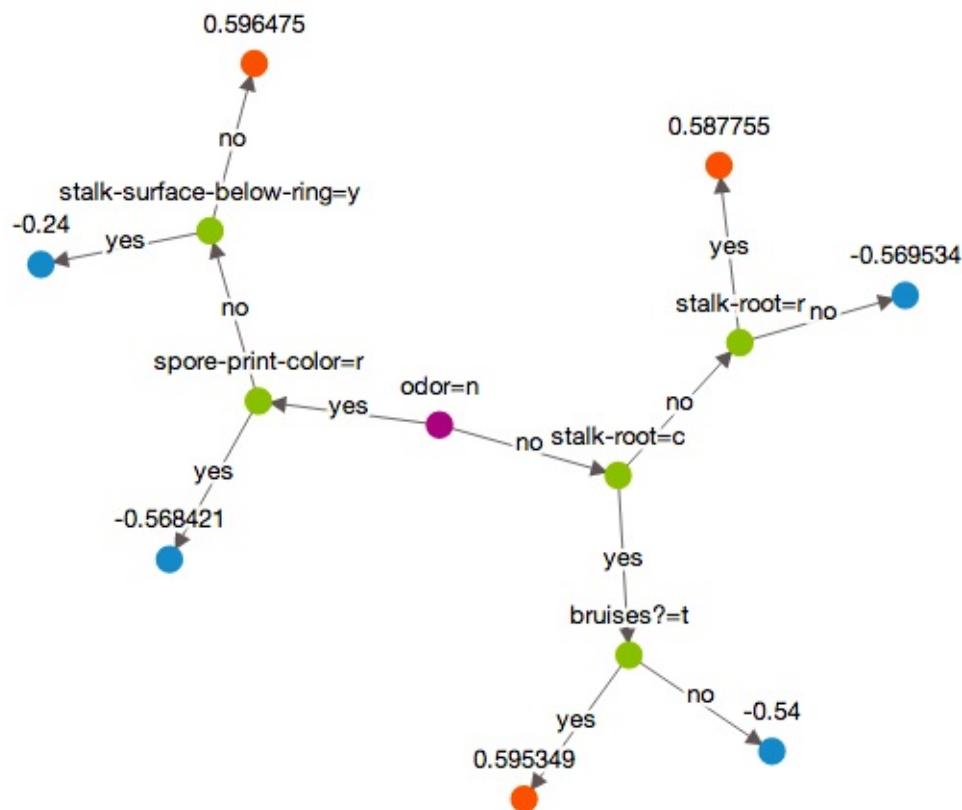
# Save predictions to an SFrame (class and corresponding class-probabilities)
predictions = model.classify(test_data)

# Evaluate the model and save the results into a dictionary
results = model.evaluate(test_data)
```

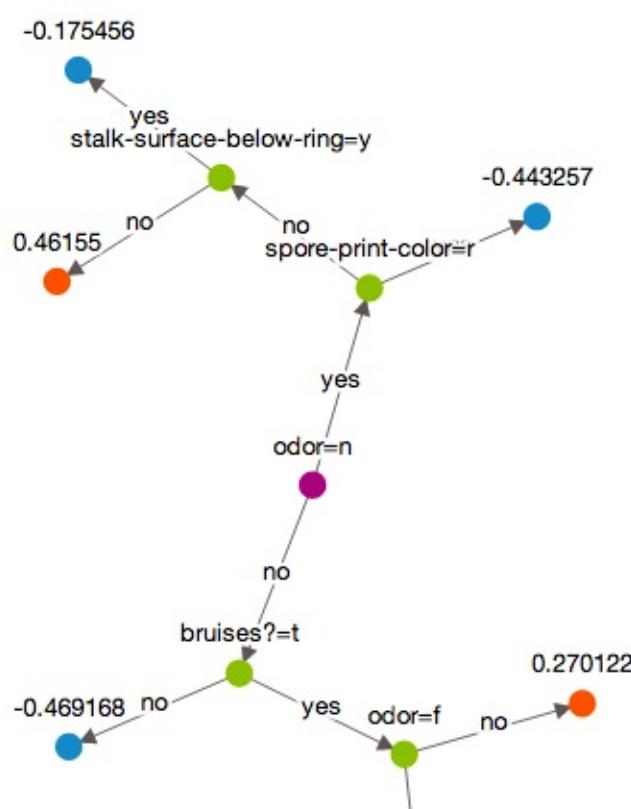
We can visualize the models using

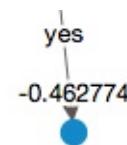
```
model.show(view="Tree", tree_id=0)
model.show(view="Tree", tree_id=1)
```

tree_0



tree_1





Tuning hyperparameters

The Gradient Boosted Trees model has many tuning parameters. Here we provide a simple guideline for tuning the model.

- `max_iterations` Controls the number of trees in the final model. Usually the more trees, the higher accuracy. However, both the training and prediction time also grows linearly in the number of trees.
- `max_depth` Restricts the depth of each individual tree to prevent overfitting.
- `step_size` Also called shrinkage, appeared as the η in the equations in the Background section. It works similar to the learning rate of the gradient descent procedure: smaller value will take more iterations to reach the same level of training error of a larger step size. So there is a trade off between `step_size` and number of iterations.
- `min_child_weight` One of the pruning criteria for decision tree construction. In classification problem, this corresponds to the minimum observations required at a leaf node. Larger value produces simpler trees.
- `min_loss_reduction` Another pruning criteria for decision tree construction. This restricts the reduction of loss function for a node split. Larger value produces simpler trees.
- `row_subsample` Use only a fraction of data at each iteration. This is similar to the mini-batch [stochastic gradient descent](#) which not only reduce the computation cost of each iteration, but may also produce more robust model.
- `column_subsample` Use only a subset of the columns to use at each iteration.

See the chapter on [boosted trees regression](#) for additional tips and tricks of using the boosted trees classifier model.

Advanced Features

Refer to the earlier chapters for the following features:

- [Using categorical features](#)
- [Sparse features](#)
- [List features](#)
- [Evaluating Results](#)

- Multiclass Classification
- Working with imbalanced data

Neuralnet Classifier

The deep learning module is useful to create and manipulate different neural network architectures. The core of this module is the `NeuralNet` class, which stores the definition of each layer of a neural network and a dictionary of learning parameters.

A `NeuralNet` object can be obtained from `graphlab.deeplearning.create`. The function, selects a *default* network architecture depending on the input dataset using simple rules: it creates a 2-layer Perceptron Network for dense numeric input, and a 1-layer Convolution Network for image data input.

Warning: Due to the high complexity of netualnet models, the default network does not always work out of the box, and you will often need to tweak the architectures a bit to make it work for your problem.

Introductory Example: Digit Recognition on MNIST Data

In this example, we will train a covolutional neural network for digit recognition. We need to make sure all of the images are the same size, since neural nets have fixed input size. We can use the `resize` function. The setup code to get started is as follows:

```
import graphlab as gl

# Load the MNIST data (from an S3 bucket)
data = gl.SFrame('https://static.turi.com/datasets/mnist/sframe/train')
test_data = gl.SFrame('https://static.turi.com/datasets/mnist/sframe/test')

# Random split the training-data
training_data, validation_data = data.random_split(0.8)

# Make sure all images are of the same size (Required by Neuralnets)
for sf in [training_data, validation_data, test_data]:
    sf['image'] = gl.image_analysis.resize(sf['image'], 28, 28, 1)
```

We will use the builtin `NeuralNet` architecture for MNIST (a one layer convolutional neuralnet work). The layers of the network can be viewed as follows.

```

net = gl.deeplearning.get_builtin_neuralnet('mnist')

print "Layers of the network "
print "-----"
print net.layers

print "Parameters of the network "
print "-----"
print net.params

```

```

Layers of the network
-----
layer[0]: ConvolutionLayer
    padding = 1
    stride = 2
    random_type = xavier
    num_channels = 32
    kernel_size = 3
layer[1]: MaxPoolingLayer
    stride = 2
    kernel_size = 3
layer[2]: FlattenLayer
layer[3]: DropoutLayer
    threshold = 0.5
layer[4]: FullConnectionLayer
    init_sigma = 0.01
    num_hidden_units = 100
layer[5]: SigmoidLayer
layer[6]: FullConnectionLayer
    init_sigma = 0.01
    num_hidden_units = 10
layer[7]: SoftmaxLayer

Parameters of the network
-----
{'batch_size': 100,
 'divideby': 255,
 'init_random': 'gaussian',
 'l2_regularization': 0.0,
 'learning_rate': 0.1,
 'momentum': 0.9}

```

We can now train the neural network using the specified network as follows:

```

model = gl.neuralnet_classifier.create(training_data, target='label',
                                       network = net,
                                       validation_set=validation_data,
                                       metric=['accuracy', 'recall@2'],
                                       max_iterations=3)

```

Making Predictions

We can now classify the test data, and output the most likely class label. The score corresponds to the learned probability of the testing instance belonging to the predicted class.

Note that this is inherently a **multi-class** classification problem, so the `classify` provides the **top** label predictions for each data point along with a probability/confidence of the class prediction.

```
predictions = model.classify(test_data)
print predictions
```

```
+-----+-----+
| row_id | class |      score      |
+-----+-----+
|    0   |    0   | 0.998417854309 |
|    1   |    0   | 0.999230742455 |
|    2   |    0   | 0.999326109886 |
|    3   |    0   | 0.997855246067 |
|    4   |    0   | 0.997171103954 |
|    5   |    0   | 0.996235311031 |
|    6   |    0   | 0.999143242836 |
|    7   |    0   | 0.999519705772 |
|    8   |    0   | 0.999182283878 |
|    9   |    0   | 0.999905228615 |
| ...   | ...   |     ...      |
+-----+-----+
[10000 rows x 3 columns]
```

Making Detailed Predictions

We can use the `predict_topk` interface if we want prediction scores for each class in the **top-k** classes (sorted in decreasing order of score).

Predict the top 2 most likely digits

```
pred_top2 = model.predict_topk(test_data, k=2)
print pred_top2
```

```
+-----+-----+-----+
| row_id | class |      score      |
+-----+-----+-----+
|  0    |  0   |  0.998417854309 |
|  0    |  6   |  0.000686840794515 |
|  1    |  0   |  0.999230742455 |
|  1    |  2   |  0.000284609268419 |
|  2    |  0   |  0.999326109886 |
|  2    |  8   |  0.000261707202299 |
|  3    |  0   |  0.997855246067 |
|  3    |  8   |  0.00118813838344 |
|  4    |  0   |  0.997171103954 |
|  4    |  6   |  0.00115600414574 |
| ...  | ...  |     ...      |
+-----+-----+-----+
[20000 rows x 3 columns]
```

Evaluating the Model

We can evaluate the classifier on the test data. Default metrics are accuracy, and confusion matrix.

```
result = model.evaluate(test_data)
print "Accuracy      : %s" % result['accuracy']
print "Confusion Matrix : \n%s" % result['confusion_matrix']
```

```
Accuracy      : 0.977599978447
Confusion Matrix :
+-----+-----+-----+
| target_label | predicted_label | count |
+-----+-----+-----+
|  0    |  0   |  973  |
|  2    |  0   |   4   |
|  4    |  0   |   1   |
|  5    |  0   |   2   |
|  6    |  0   |   9   |
|  7    |  0   |   1   |
|  8    |  0   |   1   |
|  9    |  0   |   3   |
|  1    |  1   | 1122  |
|  2    |  1   |   1   |
| ...  | ...  |     ... |
+-----+-----+-----+
[65 rows x 3 columns]
Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.
```

Using a Neural Network for Feature Extraction

A previously trained model can be used to extract dense features for a given input. The `extract_features` function takes an input dataset, propagates each example through the network, and returns an SArray of dense feature vectors, each of which is the concatenation of all the hidden unit values at `layer[layer_id]`. These feature vectors can be used as input to train another classifier such as a `LogisticClassifier`, an `SVMClassifier`, another `NeuralNetClassifier`, or `BoostedTreesClassifier`. Input dataset size must be the same as for the training of the model, except for images which are automatically resized.

If the original network is trained on a large dataset, these deep features can be very powerful. This is especially true in the context of image analysis, where a model trained on the very large ImageNet dataset can learn [general purpose features](#).

In this example, we will build a neural network for classification of digits, then build a generic classifier on top of those extracted features.

```
# The data is the MNIST digit recognition dataset
data = graphlab.SFrame('https://static.turi.com/datasets/mnist/sframe/train6k')
net = graphlab.deeplearning.get_builtin_neuralnet('mnist')
m = graphlab.neuralnet_classifier.create(data,
                                         target='label',
                                         network=net,
                                         max_iterations=3)

# Now, let's extract features from the last layer
data['features'] = m.extract_features(data)
# Now, let's build a new classifier on top of extracted features
m = graphlab.classifier.create(data,
                                 features = ['features'],
                                 target='label')
```

We also provide a [model trained on Imagenet](#). This pre-trained model gives pre-trained features of excellent quality for images, and the way you would use such a model is demonstrated below:

```
imagenet_path = 'https://static.turi.com/models/imagenet_model_iter45'
imagenet_model = graphlab.load_model(imagenet_path)
data['image'] = graphlab.image_analysis.resize(data['image'], 256, 256, 3)
data['imagenet_features'] = imagenet_model.extract_features(data)
```

One can also use our [feature engineering](#) tools for extracting deep features.

Regression **Regression** is the problem of learning a functional relationship between **input features** and an **output target** using training data where the specific functional form learned depends on the choice of model. The parameters of the function are learned using data where the target values are known, so that the machine can make predictions about data where the target is unknown. The goal of a regression model is to learn to **predict** an output based on an input set of features. The following figure depicts a sample supervised learning model where the training data (depicted in blue) is used to learn a function (depicted in pink) which can then be used in the future to make predictions. Creating regression models is easy with GraphLab Create! The regression toolkit implements the following models:

- [Linear regression](#)
- [Boosted Decision Trees](#)

These algorithms differ in how they make predictions, but conform to the same API. With all models, call **create()** to create a model, **predict()** to make predictions on the returned model, and **evaluate()** to measure performance of the predictions. All models can incorporate:

- Numeric features
- Categorical variables
- Sparse features (i.e feature sets that have a large set of features, of which only a small subset of values are non-zero)
- Dense features (i.e feature sets with a large number of numeric features)
- Text data
- Images

Model Selector

It isn't always clear that we know exactly which model is suitable for a given task. GraphLab Create's model selector automatically picks the right model for you based on statistics collected from the data set.

```
import graphlab as gl

# Load the data
data = gl.SFrame('https://static.turi.com/datasets/regression/yelp-data.csv')

# Make a train-test split
train_data, test_data = data.random_split(0.8)

# Automatically picks the right model based on your data.
model = gl.regression.create(train_data, target='stars',
                             features = ['user_avg_stars',
                                         'business_avg_stars',
                                         'user_review_count',
                                         'business_review_count'])

# Save predictions to an SArray
predictions = model.predict(test_data)

# Evaluate the model and save the results into a dictionary
results = model.evaluate(test_data)
```

GraphLab Create implementations are built to work with up to billions of examples and up to millions of features.

Linear Regression

GraphLab's [linear regression](#) module is used to predict a continuous **target** as a linear function of **features**. This is a two-stage process, analogous to many other GraphLab toolkits. First a model is created (or trained), using **training data**. Once the model is created, it can then be used to make predictions on new examples that were not seen in training (the **test data**). Model creation, prediction, and evaluation work will data that is contained in an SFrame. The following figure illustrates how linear regression works. Notice that the functional form learned here is a linear function (unlike the previous figure where the predicted function was non-linear).

Background

Given a set of features x_i , and a label y_i (where i denotes a single example in the training data), linear regression assumes that the target y_i is a linear combination of the features x_i i.e

$$y_i = f_i(\theta) = \theta^T x + \epsilon_i$$

where ϵ_i is random Gaussian noise with mean 0 and variance σ . An **intercept term** is added by appending a column of 1's to the features. Regularization is often required to prevent over-fitting by penalizing models with extreme parameter values. The linear regression module supports L1 and L2 [regularization](#), which are added to the loss function.

The composite objective being optimized for is the following:

$$\min_{\theta} \sum_{i=1}^n (\theta^T x - y_i)^2 + \lambda_1 \|\theta\|_1 + \lambda_2 \|\theta\|_2^2$$

where λ_1 is the `l1_penalty` and λ_2 is the `l2_penalty`.

Introductory Example

In this chapter, we will use a sampled version of the data from the [Yelp Dataset Challenge](#). The task is to **predict the 'star rating' for a restaurant for a given user**. The entire dataset is available on the [Yelp website](#).

We will first split the data into a train-test split and then create a linear regression model that can predict the **star rating** for each review using:

- Average rating of a given business
- Average rating made by a user

- Number of reviews made by a user
- Number of reviews that concern a business

```
import graphlab as gl

# Load the data
data = gl.SFrame('https://static.turi.com/datasets/regression/yelp-data.csv')

# Make a train-test split
train_data, test_data = data.random_split(0.8)

# Create a model.
model = gl.linear_regression.create(train_data, target='stars',
                                    features = ['user_avg_stars',
                                                'business_avg_stars',
                                                'user_review_count',
                                                'business_review_count'])

# Save predictions to an SFrame (class and corresponding class-probabilities)
predictions = model.predict(test_data)

# Evaluate the model and save the results into a dictionary
results = model.evaluate(test_data)
```

Additional Features

We will now go over some more advanced options with the linear regression module. This includes regularization, evaluation options, model interpretation, and missing value handling. Note that **logistic regression and support vector machines (SVM)** conform to **almost** all of the API discussed below.

Accessing attributes of the model

The attributes of all GraphLab Create models, which include training statistics, model hyper-parameters, and model results can be accessed in the same way as python dictionaries. To get a list of all fields that can be accessed, you can use the [list_fields\(\)](#) function:

```
fields = model.list_fields()
print fields
```

```
['auto_tuning',
 'coefficients',
 'convergence_threshold',
 'feature_rescaling',
 'features',
 'l1_penalty',
 'l2_penalty',
 'lbfgs_memory_level',
 'max_iterations',
 'mini_batch_size',
 'num_coefficients',
 'num_examples',
 'num_features',
 'num_unpacked_features',
 'solver',
 'step_size',
 'target',
 'training_iterations',
 'training_loss',
 'training_rmse',
 'training_solver_status',
 'training_time',
 'unpacked_features']
```

Each of these fields can be accessed using dictionary-like lookups. For example, the `training_rmse` measures the root-mean-squared error during training. It can be accessed as follows:

```
print model['training_rmse']
```

```
0.971003765928
```

The [API docs](#) provide a detailed description of each of the model's attributes.

Interpreting results

Linear regression can provide valuable insight about the relationships between the target and feature columns in your data, revealing why your model returns the predictions that it does. The **coefficients** (θ) are what the training procedure learns. Each model coefficient describes the expected change in the target variable associated with a unit change in the feature. The bias term indicates the "inherent" or "average" target value if all feature values were set to zero.

The coefficients often tell an interesting story of how much each feature matters in predicting target values. The magnitude (absolute value) of the coefficient for each feature indicates the strength of the feature's association to the target variable, **holding all other features constant**. The sign on the coefficient (positive or negative) gives the direction of the association.

When a GraphLab Create regression model is trained, the `model.summary()` output shows the largest positive and negative coefficients. For a trained model, we can access the coefficients as follows:

```
coefs = model['coefficients']
print coefs
```

name	index	value	stderr
(intercept)	None	-2.22993154648	0.0199750718984
user_avg_stars	None	0.809669227421	0.00414406824672
business_avg_stars	None	0.781099022798	0.00419064940637
user_review_count	None	1.85023282568e-05	1.15481783547e-05
business_review_count	None	7.06842770902e-05	1.72332966093e-05

[5 rows x 3 columns]

Note that the `index` column in the coefficients is only applicable for categorical features, lists, and dictionaries. In the SFrame above, there is an extra column for `standard errors` on the estimated coefficients (see section below for a more detailed explanation).

Standard-errors

The standard error is the empirical standard deviation on the estimated coefficient. For example, a coefficient of 1 with a standard error of 0.5 implies a standard deviation of 0.5 on the estimated values of the coefficients. Standard errors capture the `confidence` we have in the coefficients estimated during model training. Smaller standard errors implies more confidence in the value of the coefficients returned by the model.

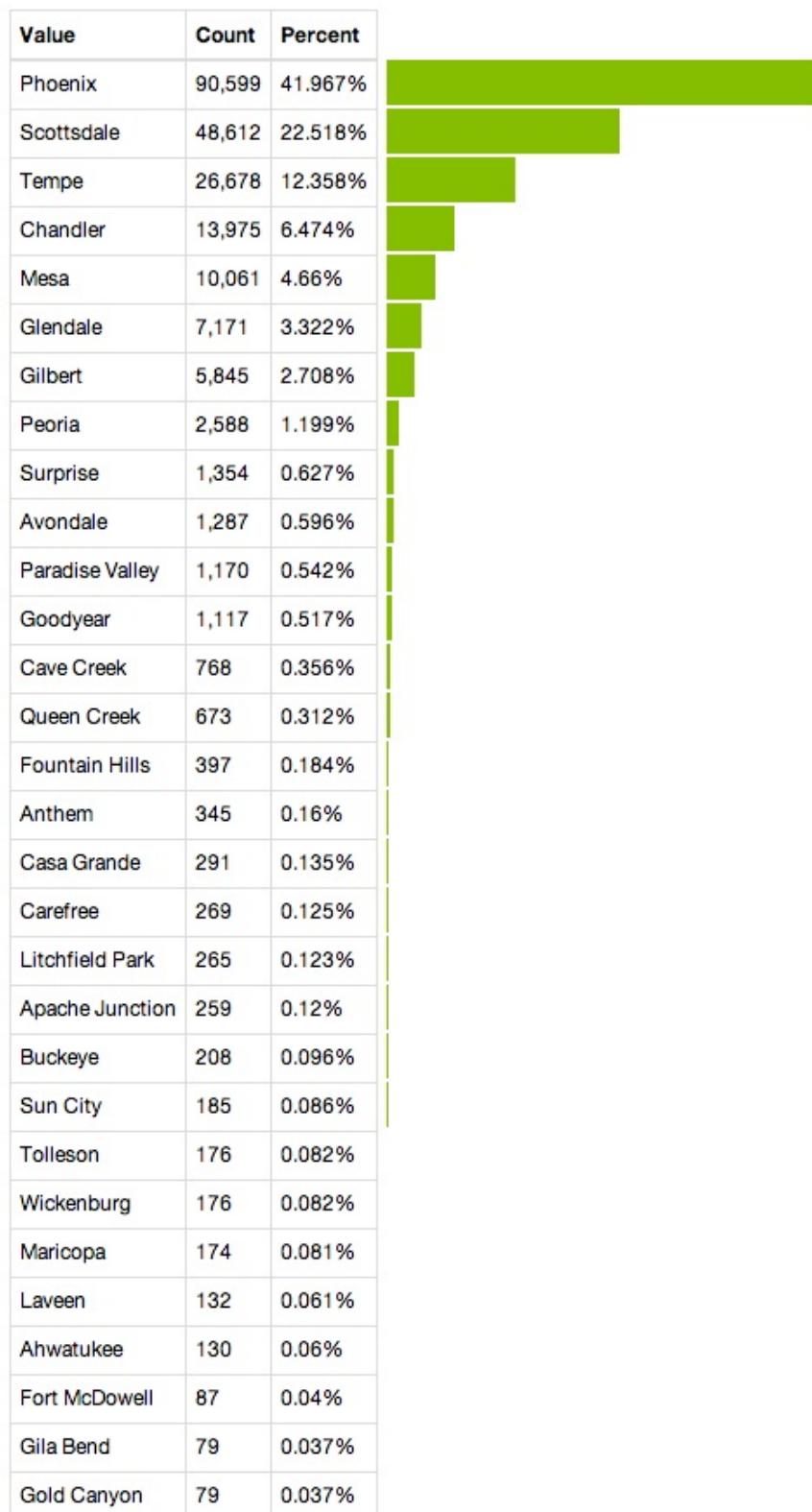
Standard errors on coefficients are only available when `solver=newton` or when the default `auto` solver option chooses the newton method and if the number of examples in the training data is more than the number of coefficients. If standard errors cannot be estimated, a column of `None` values are returned.

Categorical features

Categorical variables are features that can take one of a limited, and usually fixed, number of possible values. Each category is referred to as a level. As an example, consider the variable **city** that a given restaurant is in. This dataset has about 60 unique strings for the **city** (the number of unique values in the output of GraphLab Canvas is approximate).

```
city = train_data['city']
city.show()
```

Most frequent items from city



The regression module in Graphlab Create uses **simple encoding** while training models using string features. Simple encoding compares each category to an arbitrary reference category (we choose the first category of the data as the reference). In other words, we add *dummy coefficients* to encode each category. The number of these dummy coefficients is

equal to the total number of categories minus 1 (for the reference category). The following figure illustrates how categorical variables are encoded using the **simple encoding** scheme. [This article](#) provides more details about simple encoding for regression models.

CATEGORY	ENCODING	
	California	0 0  Reference category
	Washington	0 1
	Wisconsin	1 0

Now let us look at a simple example of using **city** as a variable in the linear regression model. Notice that there is no need to do any special pre-processing. All SFrame columns of type **str** are automatically transformed into categorical variables. Notice that the **number of coefficients** and the **number of features** aren't the same.

In this example, there are 60 unique cities, so 59 dummy coefficients. Graphlab Create can handle categorical variables with **millions** of categories.

```
# Create a model with categorical features.
model = gl.linear_regression.create(train_data, target='stars',
                                    features = ['user_avg_stars',
                                                'business_avg_stars',
                                                'user_review_count',
                                                'business_review_count',
                                                'city'])

# Number of feature columns
print "Number of features      : %s" % model['num_features']

# Number of features (including expanded lists and dictionaries)
print "Number of unpacked features : %s" % model['num_unpacked_features']

# Number of coefficients in the model
print "Number of coefficients     : %s" % model['num_coefficients']

# A coefficient is included for each category
print model['coefficients']
```

```

Number of features      : 5
Number of unpacked features : 5
Number of coefficients    : 64

+-----+-----+-----+
|       name      | index |      value   |
+-----+-----+-----+
| (intercept)    | None  | -2.23521785747 |
| user_avg_stars | None  | 0.812827992012  |
| business_avg_stars | None  | 0.778447236096  |
| user_review_count | None  | 2.48618619187e-05 |
| business_review_count | None  | 0.000102740183746 |
| city           | Tempe  | -0.0214425060978 |
| city           | Scottsdale | -0.00858553630484 |
| city           | Mesa    | 0.0109508448699  |
| city           | Chandler | 0.0390579171569  |
| city           | Gilbert  | -0.00525039510189 |
| ...            | ...     | ...          |
+-----+-----+-----+
[64 rows x 3 columns]

```

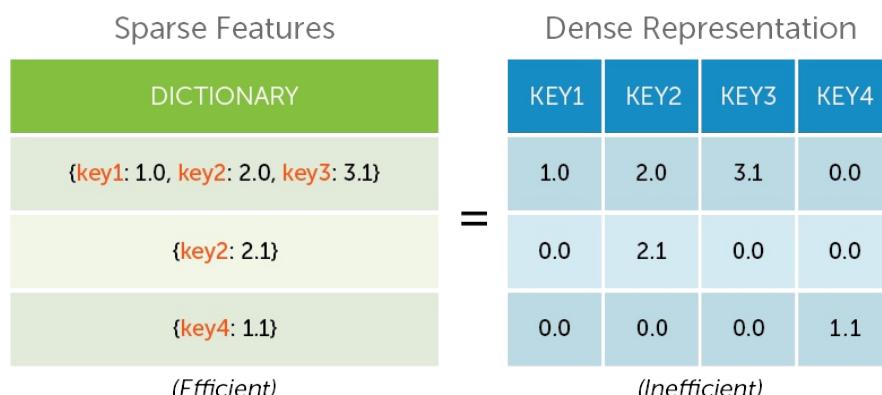
Note: Only the head of the SFrame is printed.

You can use `print_rows(num_rows=m, num_columns=n)` to print more rows and columns.

Sparse features

Sparsity is one of the most important things to consider when working with a lot of data.

Sparse features are encoded using a dictionary where the keys represent the feature names and the values are the feature values. A coefficient is added for each key in the union of the set of keys for all dictionaries across all examples. For a particular example, only non-zero values need to be stored in the dictionary. The following illustration describes how GraphLab Create dictionary features leverage sparsity in your data.



Often, the sparse representation using dictionaries is much more efficient than using the dense representation.

Yelp reviews contain useful tags that describe each business. Unlike categorical variables, a business may have several tags. These tags are stored using dictionaries. **Not all tags** are associated with every restaurant, so we will only store those tags that are associated with restaurant and assume that all tags **not explicitly provided are treated as features with value zero**. We illustrate with the following example:

```
print train_data['categories_dict'].head(3)
[{'Breakfast & Brunch': 1, 'Restaurants': 1},
 {'Restaurants': 1, 'Pizza': 1, 'Italian': 1},
 {'Dog Parks': 1, 'Parks': 1, 'Active Life': 1}]
```

We can now create a linear regression model with **categories_dict** included in the feature list. A coefficient is added for each key encountered in the input data. If the test data contains keys that were not present during training, they are silently ignored.

```
# Create a model with dictionary features.
model = gl.linear_regression.create(train_data, target='stars',
                                    features = ['user_avg_stars',
                                                'business_avg_stars',
                                                'user_review_count',
                                                'business_review_count',
                                                'categories_dict'])

# Number of feature columns
print "Number of features      : %s" % model['num_features']

# Number of features (including expanded lists and dictionaries)
print "Number of unpacked features : %s" % model['num_unpacked_features']

# Number of coefficients in the model
print "Number of coefficients     : %s" % model['num_coefficients']

# A coefficient is included for each key in the dictionary
print model['coefficients']
```

```

Number of features      : 5
Number of unpacked features : 511
Number of coefficients    : 512

+-----+-----+-----+
|       name      |   index   |     value   |
+-----+-----+-----+
| (intercept) | None     | 0.872679609626 |
| user_avg_stars | None     | 0.415478142682 |
| business_avg_stars | None     | 0.363118585079 |
| user_review_count | None     | -1.13051967735e-05 |
| business_review_count | None     | 0.000205358202417 |
| categories_dict | Breakfast & Brunch | -0.120010958039 |
| categories_dict | Restaurants | 0.271655570815 |
| categories_dict | Middle Eastern | -0.229361372902 |
| categories_dict | Dog Parks | -0.509799508435 |
| categories_dict | Parks | -0.0565527745702 |
| ... | ... | ... |
+-----+-----+-----+
[512 rows x 3 columns]
Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.

```

List features

GraphLab Create can also handle list of numerical values as features without preprocessing. The following illustration shows that the numeric-list feature is equivalent to adding several columns for each individual feature. This is especially useful for image data where the number of features generated are large and may not have natural interpretations. The following figure illustrates this idea.

Dense Features		Equivalent Representation			
	LIST	0	1	2	3
	[1.0, 2.4, 5.1, 3.7]	1.0	2.4	5.1	3.7
=	[2.0, 5.1, 7.2, 2.1]	2.0	5.1	7.2	2.1
	[3.0, 5.8, 8.3, 4.8]	3.0	5.8	8.3	4.8

As an example, we convert the **votes** (which stores the number of *cool*, *funny*, and *useful* votes for each review) column into a list of features for each vote type.

```

from array import array

# List of features
train_data['votes_list'] = train_data['votes'].apply(lambda x: x.values())
print train_data['votes_list'].head(3)
[array('d', [0.0, 5.0, 2.0]),
 array('d', [0.0, 0.0, 0.0]),
 array('d', [0.0, 2.0, 1.0])]

```

Notice that the SArray of numeric lists must be of dtype **array** and the array in each row must be **of the same length**. The `astype()` function of the SArray can help convert your data from type `array` to `list`.

```

# Create a model with numeric-list features.
model = gl.linear_regression.create(train_data, target='stars',
                                    features = ['user_avg_stars',
                                                'business_avg_stars',
                                                'user_review_count',
                                                'business_review_count',
                                                'votes_list'])

# Number of feature columns
print "Number of features      : %s" % model['num_features']

# Number of features (including expanded lists and dictionaries)
print "Number of unpacked features : %s" % model['num_unpacked_features']

# Number of coefficients in the model
print "Number of coefficients     : %s" % model['num_coefficients']

# A coefficient is included for each index in the list
print model['coefficients']

```

```

Number of features      : 5
Number of unpacked features : 7
Number of coefficients    : 8

+-----+-----+-----+
|       name        | index |   value   |
+-----+-----+-----+
| (intercept)     | None  | -2.02163219079 |
| user_avg_stars  | None  | 0.792122828319 |
| business_avg_stars | None  | 0.756871316569 |
| user_review_count | None  | 3.46394815858e-06 |
| business_review_count | None  | 7.55781795366e-05 |
| votes_list       | 0     | -0.0919094290658 |
| votes_list       | 1     | -0.113808771262 |
| votes_list       | 2     | 0.201216924886 |
+-----+-----+-----+
[8 rows x 3 columns]

```

Feature rescaling

Feature rescaling is the process of scaling individual features to be of the same scale. This process is particularly useful when features vary widely in their ranges. It is well known that **feature rescaling can make an enormous impact on accuracy**. By default, GraphLab Create rescales all the features to make sure that they have the same L2-norm. The data is **not centered** to make each column of mean zero. Categorical features, lists, and sparse features (dictionaries), are also rescaled columnwise. The feature rescaling can be disabled (**only recommended for advanced users**) by setting `feature_rescaling = False`. **The coefficients are scaled back to the original scale of the problem.**

```

model = gl.linear_regression.create(train_data, target='stars',
                                    features = ['user_avg_stars',
                                                'business_avg_stars',
                                                'user_review_count',
                                                'business_review_count'],
                                    feature_rescaling = False)

```

Choosing the solver

The optimization used to train the model is automatically picked based on the input data from a carefully engineered collection of methods. The newton method works best for datasets with plenty of examples and few features (long datasets). Limited memory BFGS (`lbfgs`) is a robust solver for wide datasets (i.e datasets with many coefficients). `fista` is the default solver for L1-regularized linear regression. Gradient-descent (GD) is another well tuned method that can work really well on L1-regularized problems. The **solvers are all automatically tuned and the default options should function well**. Solver options can,

however, be changed. In this example, we will increase `max_iterations` to 20 iterations. This is useful when `model['solver_status']` is `TERMINATED: Iteration limit reached` and the accuracy of the model is not at a desired level.

```
# Provide solver options in a dictionary
model = gl.linear_regression.create(train_data, target='stars',
                                    features = ['user_id',
                                                'business_id',
                                                'user_avg_stars',
                                                'business_avg_stars'],
                                    solver = 'lbfgs',
                                    max_iterations = 20)
```

What happens when the default solver fails?

Whenever you have convergence trouble and the number of features is < 5000, then `solver = newton` is a good choice. The Newton method is one of the most stable optimization methods and works quite well if you have few coefficients. The downside of the Newton method is its large computational complexity per- iteration.

Solver options guide

Most solvers are all automatically tuned and the default options should function well. Changing the default settings may help in some cases.

- `step_size` : The starting step size to use for the `fista` and `gd` solvers. The default is set to 1.0, this is an aggressive setting. If the first iteration is taking a considerable amount of time, then reducing the step size may speed up model training. If the step size of 1.0 is accepted in the first iteration, then consider increasing this step size to a more aggressive value.
- `max_iterations` : The maximum number of passes through the data allowed. Increasing this can result in a more accurately trained model. Consider increasing this (the default value is 10) if the training accuracy is low and the *Grad-Norm* in the display is large.
- `convergence_threshold` : Convergence is tested using variation in the training objective. The variation in the training objective is calculated by dividing the difference between the objective values between two steps. Consider reducing this below the default value (0.01) for a better training data fit. Beware of *overfitting* (i.e a model that works well only on the training data) if this parameter is set to a very low value.
- `lbfgs_memory_level` : The L-BFGS algorithm keeps track of gradient information from the previous `lbfgs_memory_level` iterations. The storage requirement for each of these gradients is the `num_coefficients` in the problem. Increasing the

`lbfgs_memory_level` can help improve the quality of the model trained at the cost of more memory. Setting this to more than `max_iterations` has the same effect as setting it to `max_iterations`.

Making predictions with missing data

Once the model is created and ready to make predictions, it is quite common that the input data to make predictions on are noisy. GraphLab Create can recognize the following types of missing data (stored as **None**). Note that the following options only apply to **missing values during prediction or evaluation**. Missing values during model creation raise an error.

- **Missing values in individual features:** Individual features including numerical, categorical, list of numeric, and dictionary features may contain missing data. The option `missing_value_action` determines the way in which these features are handled. The **default value** value is `missing_value_action='impute'` which imputes each of the features to the mean value observed during training. Setting `missing_value_action='error'` throws an error if any missing values are encountered during prediction.
- **Missing feature columns:** Sometimes, an entire column is missing while making predictions. In this case, `missing_value_action='impute'` imputes the mean value observed during training for each row in the input data. Again, setting `missing_value_action='error'` throws an error if an entire feature column is missing during prediction.
- **New categories in categorical features:** New categories are often observed during predict time. These new categories are treated to be the same as the reference category thus having no effect on the model prediction.
- **New keys in dictionary features:** New keys in dictionary features during predict time are silently ignored and do not effect the prediction made.

```

# Create a model with categorical and dictionary features.
model = gl.linear_regression.create(train_data, target='stars',
                                    features = ['user_avg_stars',
                                                'business_avg_stars',
                                                'user_review_count',
                                                'business_review_count',
                                                'city',
                                                'categories_dict'])

# An SFrame with a single row with missing values in the 'user_avg_stars'
# predict() will impute the entire feature to the mean value observed during training
new_data = gl.SFrame({'user_avg_stars': [None],
                      'business_avg_stars': [30.0],
                      'user_review_count': [10],
                      'business_review_count': [20],
                      'city': ['Tuscon'],
                      'categories_dict': [{('Pizza'): 1}]})
prediction = model.predict(new_data)
print prediction

# An SFrame without the 'user_avg_stars' feature. predict() will impute the
# entire feature to the mean value observed during training.
new_data = gl.SFrame({'business_avg_stars': [30.0],
                      'user_review_count': [10],
                      'business_review_count': [20],
                      'city': ['Tuscon'],
                      'categories_dict': [{('Pizza'): 1}]})
prediction = model.predict(new_data)

```

Regularizing Models (Lasso, Ridge, and Elastic Net regression)

Regularization is the process of using problem structure to solve ill-posed problems or to prevent overfitting. The structure is imposed by adding a penalty term to the objective function. The linear regression module supports L1 and L2 regularization, which are added to the loss function.

As discussed above, the composite objective being optimized for is the following:

$$\min_{\theta} \sum_{i=1}^n (\theta^T x - y_i)^2 + \lambda_1 \|\theta\|_1 + \lambda_2 \|\theta\|_2^2$$

where λ_1 is the `l1_penalty` and λ_2 is the `l2_penalty`.

```
# Lasso
model = gl.linear_regression.create(train_data, target='stars',
                                    features = ['user_avg_stars',
                                                'business_avg_stars',
                                                'user_review_count',
                                                'business_review_count'],
                                    l1_penalty = 1.0,
                                    l2_penalty = 0.0)

# Ridge regression
model = gl.linear_regression.create(train_data, target='stars',
                                    features = ['user_avg_stars',
                                                'business_avg_stars',
                                                'user_review_count',
                                                'business_review_count'],
                                    l2_penalty = 1.0)

# Elastic net regression
model = gl.linear_regression.create(train_data, target='stars',
                                    features = ['user_avg_stars',
                                                'business_avg_stars',
                                                'user_review_count',
                                                'business_review_count'],
                                    l1_penalty = 1.0,
                                    l2_penalty = 1.0)
```

Decision Tree Regression

The decision tree is a simple machine learning model for getting started with regression tasks.

Background

A decision tree is a flow-chart-like structure, where each internal (non-leaf) node denotes a test on an attribute, each branch represents the outcome of a test, and each leaf (or terminal) node holds a class label. The topmost node in a tree is the root node. (see [here](#) for more details).

Introductory Example

```
import graphlab as gl

# Load the data
data = gl.SFrame.read_csv('https://static.turi.com/datasets/xgboost/mushroom.csv')

# Label 'p' is edible
data['label'] = data['label'] == 'p'

# Make a train-test split
train_data, test_data = data.random_split(0.8)

# Create a model.
model = gl.decision_tree_regression.create(train_data, target='label',
                                             max_depth = 3)

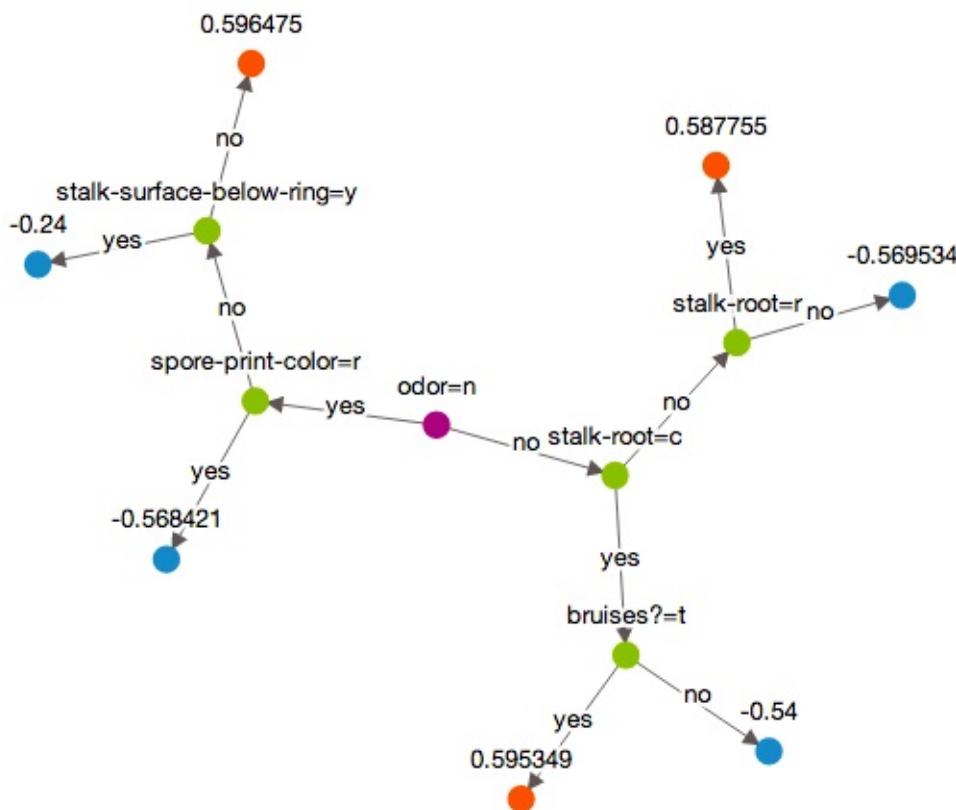
# Save predictions to an SArray
predictions = model.predict(test_data)

# Evaluate the model and save the results into a dictionary
results = model.evaluate(test_data)
```

We can visualize the models using

```
model.show(view="Tree", tree_id=0)
```

tree_0



Why chose decision trees?

Different kinds of models have different advantages. The decision tree model is very good at handling tabular data with numerical features, or categorical features with fewer than hundreds of categories. Unlike linear models, decision trees are able to capture non-linear interaction between the features and the target.

One important note is that tree based models are not designed to work with very sparse features. When dealing with sparse input data (e.g. categorical features with large dimension), we can either pre-process the sparse features to generate numerical statistics, or switch to a linear model, which is better suited for such scenarios.

Advanced Features

Refer to the earlier chapters for the following features:

- Using categorical features
 - Sparse features
 - List features
 - Evaluating Results

Gradient Boosted Regression Trees

The Gradient Boosted Regression Trees (GBRT) model (also called Gradient Boosted Machine or GBM) is one of the most effective machine learning models for predictive analytics, making it an industrial workhorse for machine learning.

Background

The Boosted Trees Model is a type of additive model that makes predictions by combining decisions from a sequence of base models. More formally we can write this class of models as:

$$g(x) = f_0(x) + f_1(x) + f_2(x) + \dots$$

where the final classifier g is the sum of simple base classifiers f_i . For boosted trees model, each base classifier is a simple [decision tree](#). This broad technique of using multiple models to obtain better predictive performance is called [model ensembling](#).

Unlike Random Forest which constructs all the base classifier independently, each using a subsample of data, GBRT uses a particular model ensembling technique called [gradient boosting](#).

The name of Gradient Boosting comes from its connection to the Gradient Descent in numerical optimization. Suppose you want to optimize a function $f(x)$, assuming f is differentiable, gradient descent works by iteratively find

$$x_{t+1} = x_t - \eta \frac{\partial f}{\partial x} \Big|_{x=x_t}$$

where η is called the step size.

Similarly, if we let $g_t(x) = \sum_{i=0}^{t-1} f_i(x)$ be the classifier trained at iteration t , and $L(y_i, g(x_i))$ be the empirical loss function, at each iteration we will move g_t towards the negative gradient direction $-\frac{\partial L}{\partial g} \Big|_{g=g_t}$ by η amount. Hence, f_t is chosen to be

$$f_t = \arg \min_f \sum_{i=1}^N \left[\frac{\partial L(y_i, g(x_i))}{\partial g(x_i)} \Big|_{g=g_t} - f(x_i) \right]^2$$

and the algorithm sets $g_{t+1} = g_t + \eta f_t$.

$$\frac{\partial L(y_i, g(x_i))}{\partial g(x_i)}$$

For regression problems with squared loss function, $\frac{\partial L(y_i, g(x_i))}{\partial g(x_i)}$ is simply $y_i - g(x_i)$. The algorithm simply fit a new decision tree to the residual at each iteration.

Introductory Example

```
import graphlab as gl

# Load the data
data = gl.SFrame.read_csv('https://static.turi.com/datasets/xgboost/mushroom.csv')

# Label 'p' is edible
data['label'] = data['label'] == 'p'

# Make a train-test split
train_data, test_data = data.random_split(0.8)

# Create a model.
model = gl.boosted_trees_regression.create(train_data, target='label',
                                             max_iterations=2,
                                             max_depth = 3)

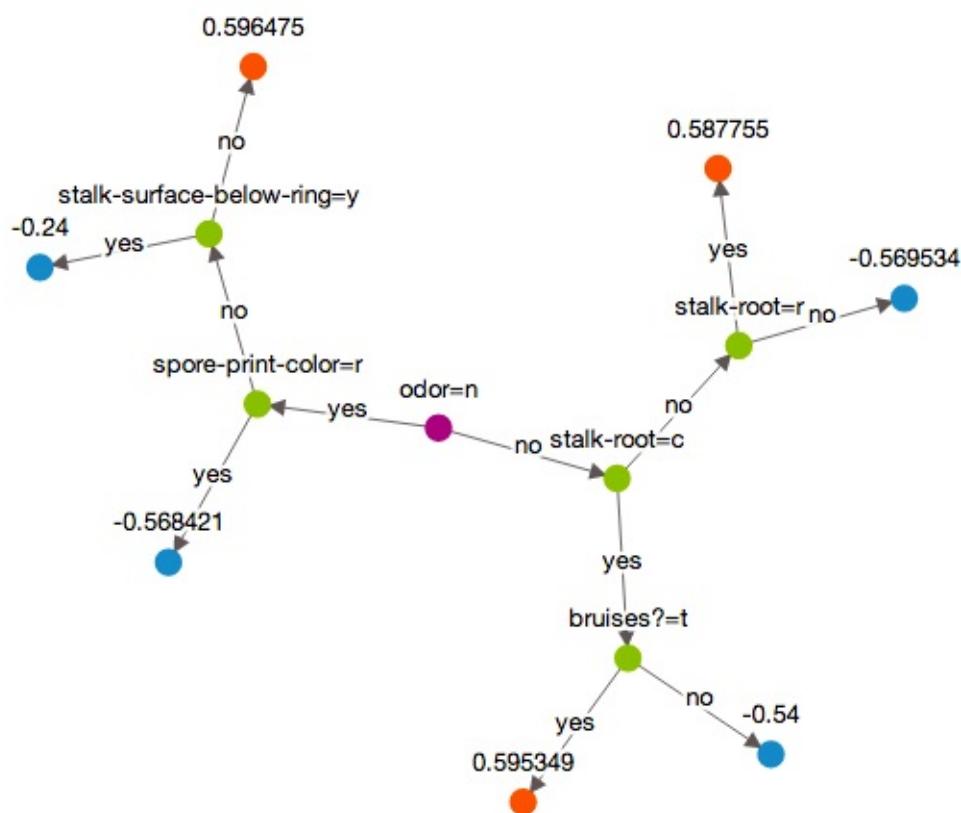
# Save predictions to an SArray
predictions = model.predict(test_data)

# Evaluate the model and save the results into a dictionary
results = model.evaluate(test_data)
```

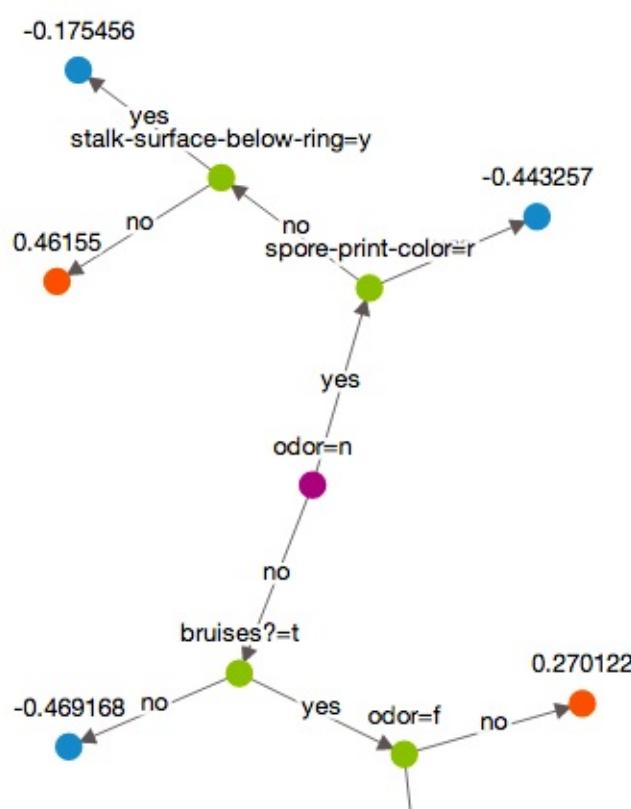
We can visualize the models using

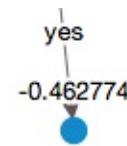
```
model.show(view="Tree", tree_id=0)
model.show(view="Tree", tree_id=1)
```

tree_0



tree_1





Tuning hyperparameters

The Gradient Boosted Trees model has many tuning parameters. Here we provide a simple guideline for tuning the model.

- `max_iterations` Controls the number of trees in the final model. Usually the more trees, the higher accuracy. However, both the training and prediction time also grows linearly in the number of trees.
- `max_depth` Restricts the depth of each individual tree to prevent overfitting.
- `step_size` Also called shrinkage, appeared as the η in the equations in the Background section. It works similar to the learning rate of the gradient descent procedure: smaller value will take more iterations to reach the same level of training error of a larger step size. So there is a trade off between `step_size` and number of iterations.
- `min_child_weight` One of the pruning criteria for decision tree construction. In classification problem, this corresponds to the minimum observations required at a leaf node. Larger value produces simpler trees.
- `min_loss_reduction` Another pruning criteria for decision tree construction. This restricts the reduction of loss function for a node split. Larger value produces simpler trees.
- `row_subsample` Use only a fraction of data at each iteration. This is similar to the mini-batch [stochastic gradient descent](#) which not only reduce the computation cost of each iteration, but may also produce more robust model.
- `column_subsample` Use only a subset of the columns to use at each iteration.

In general, you can choose `max_iterations` to be large and fit your computation budget. You can then set `min_child_weight` to be a reasonable value around (#instances/1000), and tune `max_depth`. When you have more training instances, you can set `max_depth` to a higher value. When you find a large gap between the training loss and validation loss, a sign of overfitting, you may want to reduce depth, and increase `min_child_weight`.

When to use a boosted trees model?

Different kinds of models have different advantages. The boosted trees model is very good at handling tabular data with numerical features, or categorical features with fewer than hundreds of categories. Unlike linear models, the boosted trees model are able to capture non-linear interaction between the features and the target.

One important note is that tree based models are not designed to work with very sparse features. When dealing with sparse input data (e.g. categorical features with large dimension), we can either pre-process the sparse features to generate numerical statistics, or switch to a linear model, which is better suited for such scenarios.

Random Forest Regression

The Random Forest is one of the most effective machine learning models for predictive analytics, making it an industrial workhorse for machine learning.

Background

The **random forest** model is a type of additive model that makes predictions by combining decisions from a sequence of base models. More formally we can write this class of models as:

$$g(x) = f_0(x) + f_1(x) + f_2(x) + \dots$$

where the final model g is the sum of simple base models f_i . Here, each base classifier is a simple **decision tree**. This broad technique of using multiple models to obtain better predictive performance is called **model ensembling**. In random forests, all the base models are constructed independently using a **different subsample** of the data.

Introductory Example

```
import graphlab as gl

# Load the data
data = gl.SFrame.read_csv('https://static.turi.com/datasets/xgboost/mushroom.csv')

# Label 'p' is edible
data['label'] = data['label'] == 'p'

# Make a train-test split
train_data, test_data = data.random_split(0.8)

# Create a model.
model = gl.random_forest_regression.create(train_data, target='label',
                                             max_iterations=2,
                                             max_depth = 3)

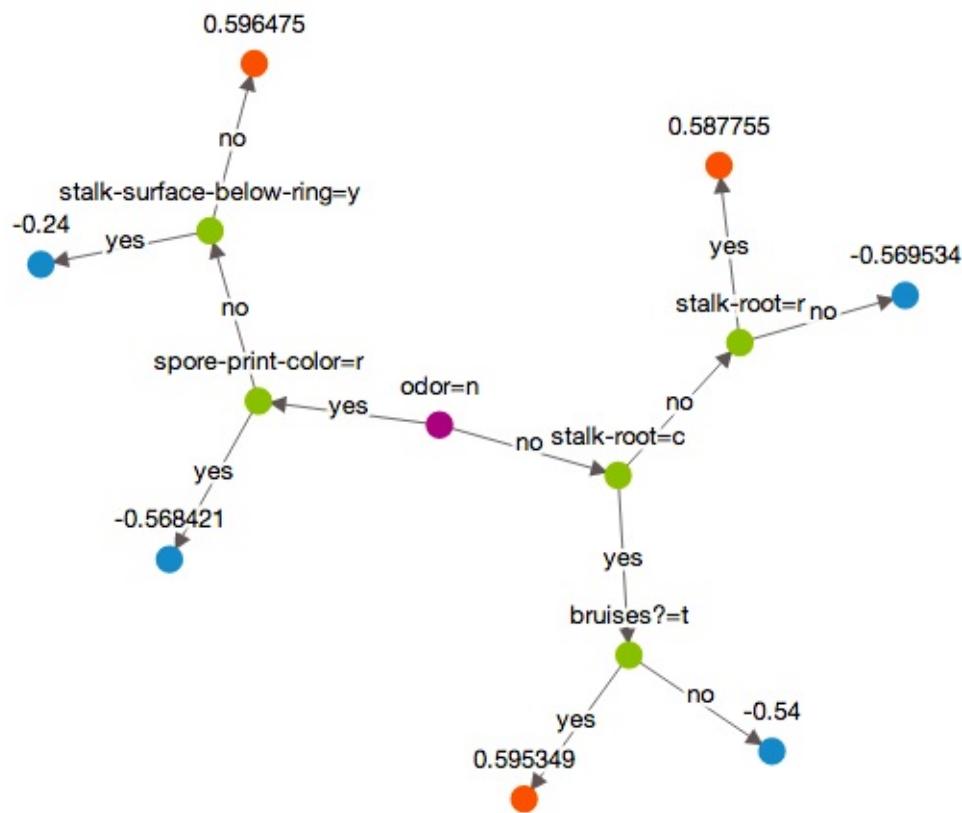
# Save predictions to an SArray
predictions = model.predict(test_data)

# Evaluate the model and save the results into a dictionary
results = model.evaluate(test_data)
```

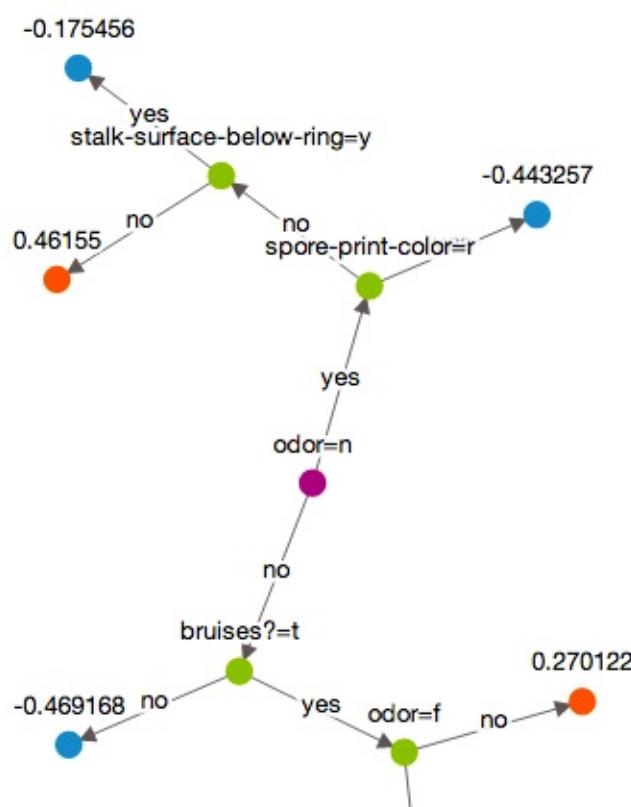
We can visualize the models using

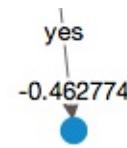
```
model.show(view="Tree", tree_id=0)
model.show(view="Tree", tree_id=1)
```

tree_0



tree_1





Tuning hyperparameters

The Gradient Boosted Trees model has many tuning parameters. Here we provide a simple guideline for tuning the model.

- **num_trees** Controls the number of trees in the final model. Usually the more trees, the higher accuracy. However, both the training and prediction time also grows linearly in the number of trees.
- **max_depth** Restricts the depth of each individual tree to prevent overfitting.
- **step_size** Also called shrinkage, appeared as the η in the equations in the background section. It works similar to the learning rate of the gradient descent procedure: smaller value will take more iterations to reach the same level of training error of a larger step size. So there is a trade off between step_size and number of iterations.
- **min_child_weight** One of the pruning criteria for decision tree construction. In classification problem, this corresponds to the minimum observations required at a leaf node. Larger value produces simpler trees.
- **min_loss_reduction** Another pruning criteria for decision tree construction. This restricts the reduction of loss function for a node split. Larger value produces simpler trees.
- **row_subsample** Use only a fraction of data at each iteration. This is similar to the mini-batch [stochastic gradient descent](#) which not only reduce the computation cost of each iteration, but may also produce more robust model.
- **column_subsample** Use only a subset of the columns to use at each iteration.

In general, you can choose `num_trees` to be as large as your computation budget permits. You can then set `min_child_weight` to be a reasonable value around (#instances/1000), and tune `max_depth`. When you have more training instances, you can set `max_depth` to a higher value. When you find a large gap between the training loss and validation loss, a sign of overfitting, you may want to reduce depth, and increase `min_child_weight`.

Why chose random forests?

Different kinds of models have different advantages. The random forest model is very good at handling tabular data with numerical features, or categorical features with fewer than hundreds of categories. Unlike linear models, random forests are able to capture non-linear interaction between the features and the target.

One important note is that tree based models are not designed to work with very sparse features. When dealing with sparse input data (e.g. categorical features with large dimension), we can either pre-process the sparse features to generate numerical statistics, or switch to a linear model, which is better suited for such scenarios.

Advanced Features

Refer to the earlier chapters for the following features:

- [Using categorical features](#)
- [Sparse features](#)
- [List features](#)
- [Evaluating Results](#)

Advanced Deep Learning with MXNet

MXNet is an open source deep learning framework designed for efficiency and flexibility. GraphLab Create integrates MXNet for creating advanced deep learning models.

MXNet makes it easy to create state-of-the-art network architectures including deep convolution neural networks (CNN), and recurrent neural networks (RNN). MXNet supports multiple CPUs and GPUs out-of-the-box: the computation is represented as symbolic graph and automatically parallelized across multiple devices. Recent benchmarks showed MXNet performed equally or faster than other frameworks such as TensorFlow, Torch or Caffe.

GraphLab Create embraces MXNet as its own module, and adds features like SFrame integration to further accelerate the creation or fine tuning your own advanced deep learning models with numerical, text, or image data.

For documentation on the MXNet design, please visit <http://mxnet.readthedocs.io/en/latest/>

Introductory Example: Linear Regression

```
import graphlab as gl
from graphlab import mxnet as mx

# Define the network symbol, equivalent to linear regression
net = mx.symbol.Variable('data')
net = mx.symbol.FullyConnected(data=net, name='fc1', num_hidden=1)
net = mx.symbol.LinearRegressionOutput(data=net, name='lr')

# Load data into SFrame and normalize features
sf = gl.SFrame.read_csv('https://static.turi.com/datasets/regression/houses.csv')
features = ['tax', 'bedroom', 'bath', 'size', 'lot']
for f in features:
    sf[f] = sf[f] - sf[f].mean()
    sf[f] = sf[f] / sf[f].std()

# Prepare the input iterator from SFrame
# `data_name` must match the first layer's name of the network.
# `label_name` must match the last layer's name plus "_label".
dataiter = mx.io.SFrameIter(sf, data_field=features, label_field='price',
                            data_name='data', label_name='lr_label',
                            batch_size=1)

# Train the network
model = mx.model.FeedForward.create(symbol=net, X=dataiter, num_epoch=20,
                                      learning_rate=1e-2,
                                      eval_metric='rmse')

# Make prediction
model.predict(dataiter)
```

Introductory Example: Logistic Regression

```

import graphlab as gl
import numpy as np
from graphlab import mxnet as mx

# Define the network symbol, equivalent to logistic regression
net = mx.symbol.Variable('data')
net = mx.symbol.FullyConnected(data=net, name='fc1', num_hidden=1)
net = mx.symbol.LinearRegressionOutput(data=net, name='lr')

# Load data into SFrame and normalize features
sf = gl.SFrame.read_csv('https://static.turi.com/datasets/regression/houses.csv')
sf['expensive'] = sf['price'] > 100000
features = ['tax', 'bedroom', 'bath', 'size', 'lot']
for f in features:
    sf[f] = sf[f] - sf[f].mean()
    sf[f] = sf[f] / sf[f].std()

# Prepare the input iterator from SFrame
# `data_name` must match the first layer's name of the network.
# `label_name` must match the last layer's name plus "_label".
dataiter = mx.io.SFrameIter(sf, data_field=features, label_field='expensive',
                            data_name='data', label_name='lr_label',
                            batch_size=1)

# Define the custom evaluation function for binary accuracy
def binary_acc(label, pred):
    return int(label[0]) == int(pred[0] >= 0.5)

model = mx.model.FeedForward.create(symbol=net, X=dataiter, num_epoch=20,
                                    learning_rate=1e-2,
                                    eval_metric=mx.metric.np(binary_acc))

# Make prediction
model.predict(dataiter)

```

Train your own Convolution Neural Network (CNN) for Image Classification

```

import graphlab as gl
from graphlab import mxnet as mx
import numpy as np

# Define the network symbol
data = mx.symbol.Variable('data')
conv1= mx.symbol.Convolution(data = data, name='conv1', num_filter=32, kernel=(3,3), stride=(2,2), pad=(1,1))
bn1 = mx.symbol.BatchNorm(data = conv1, name="bn1")
act1 = mx.symbol.Activation(data = bn1, name='relu1', act_type="relu")
mp1 = mx.symbol.Pooling(data = act1, name = 'mp1', kernel=(2,2), stride=(2,2), pool_type="max")

conv2= mx.symbol.Convolution(data = mp1, name='conv2', num_filter=32, kernel=(3,3), stride=(2,2), pad=(1,1))
bn2 = mx.symbol.BatchNorm(data = conv2, name="bn2")
act2 = mx.symbol.Activation(data = bn2, name='relu2', act_type="relu")
mp2 = mx.symbol.Pooling(data = act2, name = 'mp2', kernel=(2,2), stride=(2,2), pool_type="max")

fl = mx.symbol.Flatten(data = mp2, name="flatten")
fc2 = mx.symbol.FullyConnected(data = fl, name='fc2', num_hidden=10)
softmax = mx.symbol.SoftmaxOutput(data = fc2, name = 'sm')

# Load MINST image data into SFrame
sf = gl.SFrame('https://static.turi.com/datasets/mnist/sframe/train')

batch_size = 100
num_epoch = 1

# Prepare the input iterator from SFrame
# `data_name` must match the first layer's name of the network.
# `label_name` must match the last layer's name plus "_label".
dataiter = mx.io.SFrameImageIter(sf, data_field=['image'],
                                  label_field='label',
                                  data_name='data',
                                  label_name='sm_label', batch_size=batch_size)

# Train the network
model = mx.model.FeedForward.create(softmax, X=dataiter,
                                     num_epoch=num_epoch,
                                     learning_rate=0.1, wd=0.0001,
                                     momentum=0.9,
                                     eval_metric=mx.metric.Accuracy())

# Make prediction
model.predict(dataiter)

```

Model Creation

Model Training

As the examples above showed, `model.FeedForward.create` is the high level for training all kinds of neural networks. The main parameters are `symbol` and `x`, binding to the network architectures and the data iterator respectively. Additional parameters such as `num_epoch`, `optimizer` are used to control the optimization procedure. The default optimizer is `optimizer.SGD`. For convenience, `model.FeedForward.create` also takes optimization related parameters as `kwargs`, i.e. `learning_rate`, `momentum`. For instance:

```
model = mx.model.FeedForward.create(symbol=net, X=dataiter, learning_rate=0.01)
```

is equivalent to

```
sgd = mx.optimizer.SGD(learning_rate=0.01, rescale_grad=1.0/batch_size)
model = mx.model.FeedForward.create(symbol=net, X=dataiter, optimizer=sgd)
```

Callback functions can be used to print progress or checkpoint model during training. There are two types of callbacks: `epoch_end_callback` and `batch_end_callback`. Both are arguments to the `model.FeedForward.create` function. For instance, the following example does model check point every epoch and print progress every 10 batches.

```
model = mx.model.FeedForward.create(symbol=net, X=dataiter,
                                    batch_end_callback=mx.callback.Speedometer(batch_si
epoch_end_callback=mx.callback.do_checkpoint(prefix
```

`epoch_end_callback` is called at the end of each epoch and `batch_end_callback` is called at then end of each batch. `epoch_end_callback` types include `callback.do_checkpoint` and `callback.log_train_metric`. `batch_end_callback` types include `callback.Speedometer` and `callback.ProgressBar`.

Multiple GPU support

MXNet supports using multiple GPUs for model training and prediction. GPU support is available for Linux operating systems that have

1. Nvidia CUDA 7.0 capable GPU(s)
2. CUDA toolkit v7.0
3. Minimum driver version of 346.xx. ([Link <http://www.nvidia.com/Download/index.aspx>](http://www.nvidia.com/Download/index.aspx))

By default, the CUDA toolkit library is installed at `/usr/local/cuda`. If the CUDA toolkit is installed at a non-default location, please set the environment variable `LD_LIBRARY_PATH` to include the CUDA toolkit location. The following code shows an example that MXNet GPU capability is activated:

```
>>> from graphlab import mxnet as mx
2016-04-15 11:37:22,580 [INFO] graphlab.mxnet.base, 42: CUDA GPU support is activated
```

By default, model training and prediction uses CPU. When Nvidia CUDA-enabled GPU is available and drivers are properly installed, you can specify using a single GPU or multiple GPUs to speedup computation.

Devices in MXNet are called `context`. For example, the following code uses two GPUs for training a FeedForward network.

```
gpus = [mx.context.gpu(0), mx.context.gpu(1)]
model = mx.model.FeedForward.create(symbol=net, x=data, ctx=gpus, ...)
model2 = mx.model.FeedForward.load(..., ctx=gpus)
```

Data Input from SFrame Iterator

`graphlab.SFrame` is a scalable data frame object that can hold numericals, text, and images. Training deep neural network requires large amount of data, usually much larger than memory can hold. SFrame makes it easy to transform your large dataset and feed it to MXNet for training without the need of writing to disk with a custom file format or using a database.

MXNet integrates with `graphlab.SFrame` via `io.SFrameIter` which implements the `io.DataIter` interface. `io.SFrameIter` supports SFrame with either a single image-typed column or general tabular data with multiple numerical columns, in which the numerical columns can be of different dimensions.

`io.SFrameImageIter` is a specialized iterator for image typed data. `io.SFrameImageIter` supports image augmentation operations such as "subtracting mean pixel values" or "rescale pixel values".

These iterators can be created as follows.

SFrameImageIter:

```
# Load MINST image data into SFrame
sf = gl.SFrame('https://static.turi.com/datasets/mnist/sframe/train')

batch_size = 100
num_epoch = 1

# Prepare the input iterator from SFrame
# `data_name` must match the first layer's name of the network.
# `label_name` must match the last layer's name plus "_label".
dataiter = mx.io.SFrameImageIter(sf, data_field=['image'],
                                 label_field='label',
                                 data_name='data',
                                 label_name='sm_label', batch_size=batch_size)
```

SFramelter

```
# Load data into SFrame and normalize features
sf = gl.SFrame.read_csv('https://static.turi.com/datasets/regression/houses.csv')
features = ['tax', 'bedroom', 'bath', 'size', 'lot']
for f in features:
    sf[f] = sf[f] - sf[f].mean()
    sf[f] = sf[f] / sf[f].std()

# Prepare the input iterator from SFrame
# `data_name` must match the first layer's name of the network.
# `label_name` must match the last layer's name plus "_label".
dataiter = mx.io.SFrameIter(sf, data_field=features, label_field='price',
                            data_name='data', label_name='lr_label',
                            batch_size=1)
```

These iterators can then directly be passed to the `FeedForwardModel` creation function.

```
# Define the custom evaluation function for binary accuracy
def binary_acc(label, pred):
    return int(label[0]) == int(pred[0] >= 0.5)

model = mx.model.FeedForward.create(symbol=net, X=dataiter, num_epoch=20,
                                     learning_rate=1e-2,
                                     eval_metric=mx.metric.np(binary_acc))

# Make prediction
model.predict(dataiter)
```

Builtin Networks

The following shows the built-in state-of-the-art network architectures for image classification.

- `builtin_symbols.symbol_alexnet.get_symbol()` : Return the "AlexNet" architecture for image classification.
- `builtin_symbols.symbol_googlenet.get_symbol()` : Return the "GoogLeNet" architecture for image classification
- `builtin_symbols.symbol_vgg.get_symbol()` : Return the "VGG" architecture for image classification
- `builtin_symbols.symbol_inception_v3.get_symbol()` : Return the "Inception-v3" architecture for image classification
- `builtin_symbols.symbol_inception_bn.get_symbol()` : Return the "BN-Inception" architecture for image classification
- `builtin_symbols.symbol_inception_bn_28_small.get_symbol()` : Return a simplified version of "BN-Inception" architecture for image classification
- `builtin_symbols.symbol_inception_bn_full.get_symbol()` : Return a variant of "BN-Inception" architecture for image classification

Task Oriented Pretrained Models for Image Classification

Pretrained models are deep neural networks trained on large datasets for specific tasks. For general purpose tasks such as image classification or object detection, the quickest way to get value from deep learning is to directly apply the pretrained models.

MXNet in GLC streamlines the process of using pretrained models in the following ways:

- Provides task oriented API designed to simplify the common use cases
- Integrates with SFrame for scalable data loading and transformation
- Allows model download and management via a simple API

The following example shows the end to end process of downloading a pretrained image classifier and classifying thousands of images.

```

import graphlab as gl
from graphlab import mxnet as mx

mx.pretrained_model.download_model('https://static.turi.com/models/mxnet_models/release')

mx.pretrained_model.list_models()

image_classifier = mx.pretrained_model.load_model('imagenet1k_inception_bn', ctx=mx.gpu(0))

# Load image data into SFrame
sf = gl.SFrame('https://static.turi.com/datasets/cats_dogs_sf')

# Predict using the pretrained image classifier
prediction = image_classifier.predict_topk(sf['image'], k=1)

# Extract features from images
features = image_classifier.extract_features(sf['image'])

```

Task Oriented Pretrained Models for Object Detection

Turi also provides a pre-trained object detector. Object detection is the task of identifying objects in an image, and also providing bounding boxes for them. This is generally a challenging task, but neural networks have proven to be quite effective.

```

import graphlab as gl
from graphlab import mxnet as mx

mx.pretrained_model.download_model('https://static.turi.com/models/mxnet_models/release')

mx.pretrained_model.list_models()

image_detector = mx.pretrained_model.load_model('coco_vgg_16', ctx=mx.gpu(0))

# Load image data into SFrame
sf = gl.SFrame('https://static.turi.com/datasets/cats_dogs_sf')

# Predict using the pretrained image classifier
prediction = image_detector.detect(sf['image'][0], k=1)

image_detector.visualize_detection(sf['image'][0], prediction)

```

Pretrained Image Classifiers

InceptionBN (ImageNet ILVRC 2012)

URL:

https://static.turi.com/models/mxnet_models/release/image_classifier/imagenet1k_inception_bn-1.0.tar.gz

This model is provided by DMLC's `mxnet-model-gallery` project. More details about the training of the model can be found at [This://github.com/dmlc/mxnet-model-gallery/blob/master/imagenet-1k-inception-bn.md]

InceptionV3 (ImageNet ILVRC 2012)

URL:

https://static.turi.com/models/mxnet_models/release/image_classifier/imagenet1k_inception_v3-1.0.tar.gz

This model is provided by DMLC's `mxnet-model-gallery` project. More details about the training of the model can be found at [This://github.com/dmlc/mxnet-model-gallery/blob/master/imagenet-1k-inception-v3.md]

Inception21k (Full ImageNet)

URL:

https://static.turi.com/models/mxnet_models/release/image_classifier/imagenet21k_inception_bn-1.0.tar.gz

This model is provided by DMLC's `mxnet-model-gallery` project. More details about the training of the model can be found at [This://github.com/dmlc/mxnet-model-gallery/blob/master/imagenet-21k-inception.md]

FastNet (ImageNet ILVRC 2012)

URL:

https://static.turi.com/models/mxnet_models/release/image_classifier/imagenet21k_fastnet-1.0.tar.gz

This model is trained with a modified architecture from AlexNet with reduced number of parameters while keeping similar accuracy.

PlacesVGG (Places2 2015)

URL: https://static.turi.com/models/mxnet_models/release/image_classifier/places_vgg_16-1.0.tar.gz

This model is trained with VGG-16 architechture, on the Places2 dataset. The Places2 dataset contains 8 million images of 400 different scene categories. More details about the dataset can be found at <http://places2.csail.mit.edu/>

LeNet (MNIST)

URL:

https://static.turi.com/models/mxnet_models/release/image_classifier/mnist_lenet.tar.gz

This model is trained with the classic LeNet architechture, on the MNIST dataset. The MNIST dataset contains 70,000 images of handwritten digits. More details about the dataset can be found at <http://yann.lecun.com/exdb/mnist/>. More information about the LeNet architechture can be found at <http://yann.lecun.com/exdb/lenet/>.

Pretrained Image Detectors

COCO_VGG_16 (COCO)

URL: https://static.turi.com/models/mxnet_models/release/image_detector/coco_vgg_16-1.0.tar.gz

This model is trained with VGG-16 architechture, on the Microsoft COCO dataset. The COCO dataset is a recent dataset often used for detection. More details about the dataset can be found at <http://mscoco.org/>.

Symbols (Layers)

`symbol.Symbol` is the building block of a neural network. Every symbol can be viewed as a functional object with forward and backward operation. Individual symbols can be composed into more complexed symbol which becomes a neural network. An example is below:

```
data = mx.symbol.Variable('data')
conv1= mx.symbol.Convolution(data = data, name='conv1', num_filter=32, kernel=(3,3), stride=(2,2), pad=(1,1))
bn1 = mx.symbol.BatchNorm(data = conv1, name="bn1")
act1 = mx.symbol.Activation(data = bn1, name='relu1', act_type="relu")
mp1 = mx.symbol.Pooling(data = act1, name = 'mp1', kernel=(2,2), stride=(2,2), pool_type="avg")

conv2= mx.symbol.Convolution(data = mp1, name='conv2', num_filter=32, kernel=(3,3), stride=(2,2), pad=(1,1))
bn2 = mx.symbol.BatchNorm(data = conv2, name="bn2")
act2 = mx.symbol.Activation(data = bn2, name='relu2', act_type="relu")
mp2 = mx.symbol.Pooling(data = act2, name = 'mp2', kernel=(2,2), stride=(2,2), pool_type="avg")

fl = mx.symbol.Flatten(data = mp2, name="flatten")
fc2 = mx.symbol.FullyConnected(data = fl, name='fc2', num_hidden=10)
softmax = mx.symbol.SoftmaxOutput(data = fc2, name = 'sm')
```

For a more detailed tutorial on the Symbolic API, please see:

Debugging and Monitoring

`monitor.Monitor` is used to track outputs, weights and gradients during training. Create a `monitor.Monitor` object and pass to the `model.FeedForward.create` function to enable monitoring.

[monitor.Monitor](<https://turi.com/products/create/docs/generated/graphlab.mxnet.monitor.html>)

Extract Features from a Model

The following shows how to extract features from a given model. `model.extract_features` is an API for extracting features from a given model for data. The main parameters are `model` and `data`, for given `model.FeedForward` model and iterator of data. There is an optional parameter `top_layer`, which indicates from which layer features will be extracted; If `top_layer` is not set, it will automatically extract the second-to-last layer as features.

Note: Default `top_layer` is only correct for single output network

```
# net is a FeedForward model
# This will extract features in the second last layer (the last layer is classifier)
fea = mx.model.extract_features(model=net, data=dataiter)
```

or

```
# net is a FeedForward model
# This will extract features in layer `fc_output`
fea = mx.model.extract_features(model=net, data=dataiter, top_layer='fc1_output')
```

If a wrong `top_layer` name is given, correct candidates for `top_layer` will be given in exception.

Finetune a Model

There is a helper function for finetuning, which is `model.get_feature_symbol`. This function will generate a feature symbol from a given `FeedForward` model. Similar to extract features, an optional parameter `top_layer` is used for setting until which layer the network will be kept. If no value is set, the layer before the linear classifier will be used as features.

Note: Default `top_layer` is only correct for single output network

```
# net is a FeedForward model
# This will get symbol of the second last layer and below (the last layer is classifier)
feature_symbol = mx.model.get_feature_symbol(net)
```

or

```
# net is a FeedForward model
# This will get symbol of `fc1_output` and below
feature_symbol = mx.model.get_feature_symbol(net, top_layer='fc1_output')
```

After we get feature symbol, we can make new symbol for different task:

```
# net is a FeedForward model
# This will build a classifier symbol based on feature symbol
feature_symbol = mx.model.get_feature_symbol(net)
classifier = mx.sym.FullyConnected(data=feature, num_hidden=18, name='new_classifier')
classifier = mx.sym.SoftmaxOutput(data=classifier, name='softmax')
```

or

```
# net is a FeedForward model
# This will build a regressor symbol based on feature symbol
feature_symbol = mx.model.get_feature_symbol(net)
regressor = mx.sym.FullyConnected(data=feature, num_hidden=1, name='new_regressor')
regressor = mx.symbol.LinearRegressionOutput(data=net, name='lr')
```

Then the API `model.finetune` is used for finetuning a model. The main parameters are `symbol` and `model`, for new task network symbol and the given `model.FeedForward` model to be used for finetuning. Other parameters are same to `model.FeedForward.create`, for indicating training data, validation data and optimization.

Note: Usually, smaller learning rate is used for finetuning.

```
# classifier is the new symbol
# net is a FeedForward model
new_model = mx.model.finetune(symbol=classifier, model=net, num_epoch=2, learning_rate=
                               X=train, eval_data=val,
                               batch_end_callback=mx.callback.Speedometer(100))
```

Introduction to Graph Analytics

Going from inspiration to production with graph models requires knowledge of several of the graph's attributes: influential and outlier nodes, clusters and communities, hidden connections between nodes, and the ability to compare different graphs based on these attributes. The [Graph Analytics toolkit](#) enables this depth of understanding by providing several methods:

- [Connected components](#)
- [Graph coloring](#)
- [K-Core decomposition](#)
- [PageRank](#)
- [Single-source shortest path](#)
- [Triangle count](#)

Each method takes an input graph and returns a model object, which contains the run time, an SFrame with the desired output for each vertex, and a new graph whose vertices contain the output as an attribute.

Examples

To illustrate some of these methods, we'll use a previously created SGraph where vertices represent Wikipedia articles about US businesses and edges represent hyperlinks between articles. This data is available under the Creative Commons Attribution-ShareAlike 3.0 Unported License (more details here: <http://en.wikipedia.org/wiki/Wikipedia:Copyrights>). It can be downloaded from Turi's public datasets bucket on Amazon S3. The remaining methods are left for the reader to explore in the exercises at the end of the chapter.

```
import os

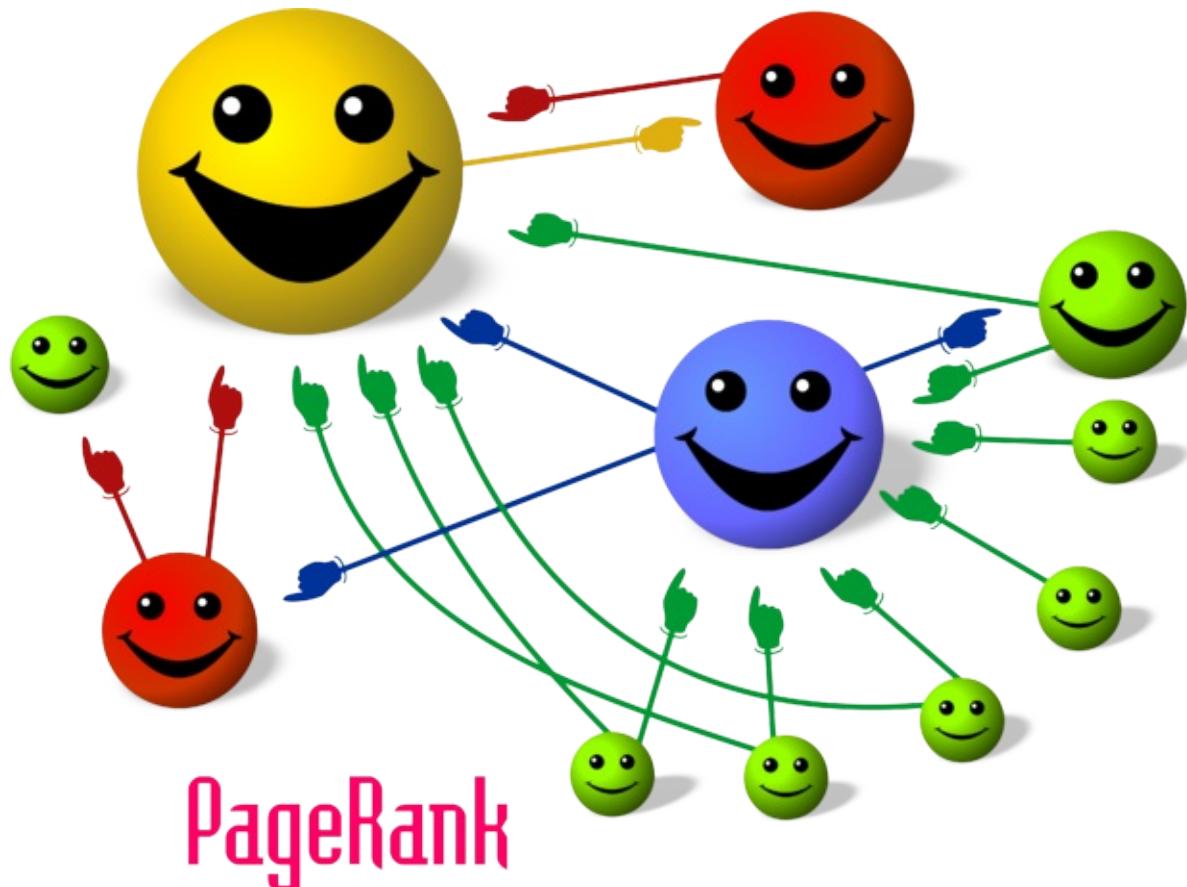
data_file = 'US_business_links'
if os.path.exists(data_file):
    sg = graphlab.load_sgraph(data_file)
else:
    url = 'https://static.turi.com/datasets/' + data_file
    sg = graphlab.load_sgraph(url)
    sg.save(data_file)

print sg.summary()
```

```
{'num_edges': 517127, 'num_vertices': 233121}
```

PageRank

PageRank is an iterative algorithm to compute the most influential nodes in a network. In each iteration, a vertex's influence is measured as the sum of influence of the nodes that point at the vertex. Each node's score is updated until the scores converge, or until the user-specified maximum number of iterations is reached.



"PageRank-hi-res". Licensed under Creative Commons Attribution-Share Alike 2.5 via Wikimedia Commons - <http://commons.wikimedia.org/wiki/File:PageRank-hi-res.png#mediaviewer/File:PageRank-hi-res.png>

As with other GraphLab Create methods, the model object is constructed with the `create` function. The `summary()` method provides a snapshot of the result, and the `list_fields()` method gives the attributes of the model that can be queried.

```
pr = graphlab.pagerank.create(sg, max_iterations=10)
print pr.summary()
```

```

PagerankModel
Graph:
+-----+-----+
| num_edges | 517127 |
| num_vertices | 233121 |
+-----+-----+
Result:
+-----+-----+
| graph | SGraph. See m['graph'] |
| pagerank | SFrame with each vertex's pagerank. See m['pagerank'] |
| delta | 3824.08346598 |
+-----+-----+
Setting:
+-----+-----+
| threshold | 0.01 |
| reset_probability | 0.15 |
| max_iterations | 10 |
+-----+-----+
Metric:
+-----+-----+
| num_iterations | 10 |
| training_time | 7.223114 |
+-----+-----+
Queriable Fields
+-----+-----+
| Field | Description |
+-----+-----+
| training_time | Total training time of the model |
| graph | A new SGraph with the pagerank as a vertex property |
| delta | Change in pagerank for the last iteration in L1 norm |
| reset_probability | The probability of randomly jumps to any node in the graph |
| pagerank | An SFrame with each vertex's pagerank |
| num_iterations | Number of iterations |
| threshold | The convergence threshold in L1 norm |
| max_iterations | The maximum number of iterations to run |
+-----+-----+

```

The model fields are retrieved either with the `get` method or by treating the model as a dictionary, as in the following snippet, which shows the model creation run time of about 7 seconds.

```
print pr['training_time']
```

```
7.234136
```

The `pagerank` field contains an SFrame with the per-node pagerank score. For this example of Wikipedia articles and hyperlinks, ABC takes the top spot by a large margin.

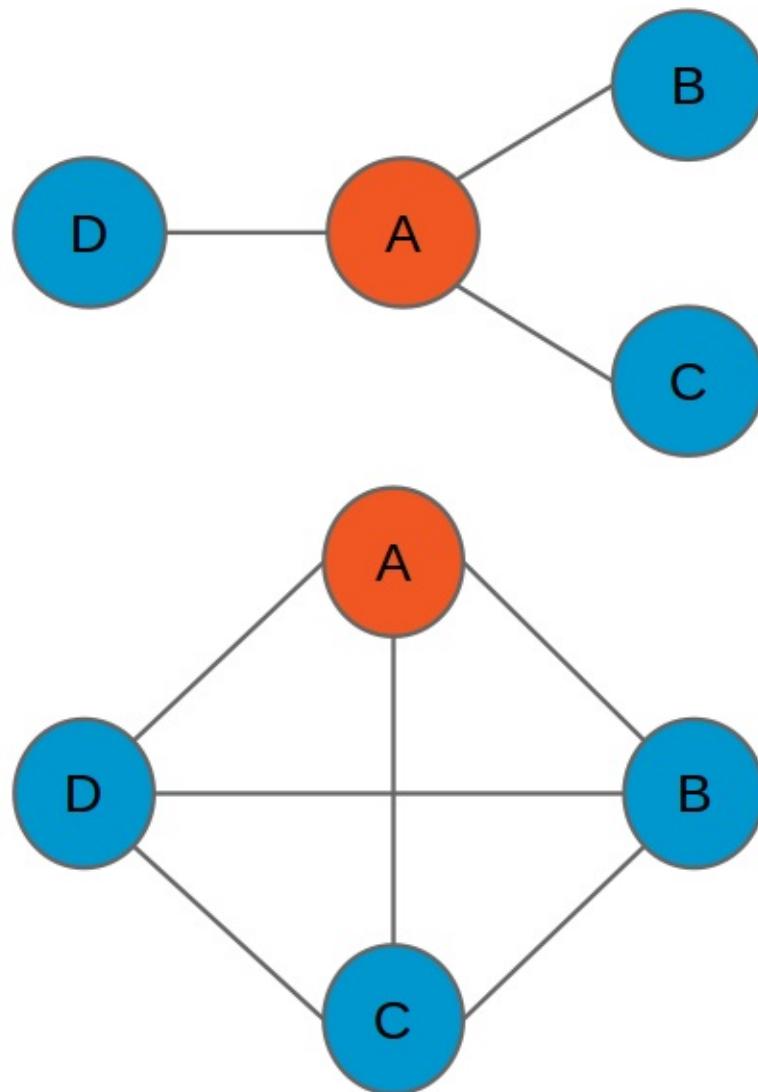
```
pr_out = pr['pagerank']
print pr_out.topk('pagerank', k=10)
```

<u>id</u>	pagerank	delta
American Broadcasting Company	3050.12285481	171.095519607
Microsoft	1640.93294801	79.4193105615
DC Comics	1623.76580168	231.05676926
Paramount Pictures	1341.29716993	74.7486642804
Facebook	1218.69295972	23.8510621222
Google	1180.48509934	41.271924661
Ford Motor Company	1156.31695182	84.5552251247
Twitter	1074.30512161	16.3932159159
Columbia Pictures	921.042912728	55.2951992749
The Walt Disney Company	878.301063411	74.9881435438

[10 rows x 3 columns]

Triangle counting

The number of triangles in a vertex's immediate neighborhood is a measure of the "density" of the vertex's neighborhood. In both of the figures below, vertex A has three immediate neighbors, ignoring edge directions. In the top figure, none of node A's neighbors is connected to any other neighbor, indicating a very loosely connected network. In contrast, all of the three neighbors are connected to each other in the bottom figure, indicating a tightly connected network.



```
tri = graphlab.triangle_counting.create(sg)
print tri.summary()
```

```

TriangleCountingModel
Graph:
+-----+-----+
| num_edges | 517127 |
| num_vertices | 233121 |
+-----+-----+
Result:
+-----+-----+
| graph | SGraph. See m['graph'] |
| num_triangles | 171968 |
| triangle_count | SFrame with each vertex's triangle count. See m['triangle_count'] |
+-----+-----+
Metric:
+-----+-----+
| training_time | 29.043092 |
+-----+-----+
Querable Fields
+-----+-----+
| Field | Description |
+-----+-----+
| graph | A new SGraph with the triangle count as a vertex property. |
| num_triangles | Total number of triangles in the graph. |
| triangle_count | An SFrame with the triangle count for each vertex. |
| training_time | Total training time of the model |
+-----+-----+

```

In this dataset, there is a lot of overlap between the companies with large influence in the Wikipedia article network (as measured by PageRank) and the companies with the largest and most dense neighborhoods, as measured by the triangle count method, but there are notable differences. ABC has only the sixth most triangles despite having the highest pagerank, while Microsoft ranks very high by both statistics, suggesting the Microsoft article is slightly more central in the network.

```

tri_out = tri['triangle_count']
print tri_out.topk('triangle_count', k=10)

```

<u>_id</u>	triangle_count
Microsoft	21447
Google	15491
Facebook	14200
IBM	11716
Paramount Pictures	10547
American Broadcasting Company	10514
Twitter	9219
Target Corporation	8474
Delta Air Lines	8195
Intel	7952

[10 rows x 2 columns]

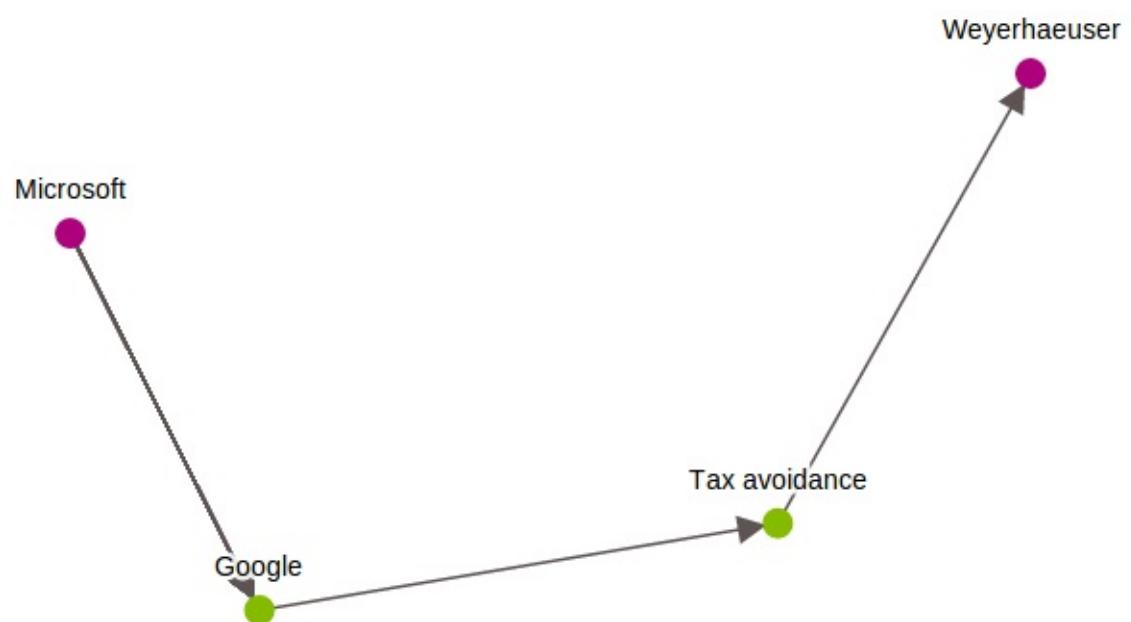
Single-source shortest path

Paths in a graph are a sequence of vertices, where each consecutive pair is connected by an edge in the graph. The single-source shortest path problem is to find the shortest path from all vertices in the graph to a user-specified target node. The source vertex with the smallest distribution of shortest paths can be considered the most central node in the graph.

Because GraphLab Create SGraphs use directed edges, the shortest path toolkit also finds the shortest directed paths to a source vertex. In this example we find all shortest paths to the node for the Microsoft article, then visualize the shortest path from the Microsoft article to the Weyerhaeuser article. Interestingly, the quickest way to get from Microsoft to Weyerhaeuser is via the articles on Google and tax avoidance.

```
sssp = graphlab.shortest_path.create(sg, source_vid='Microsoft')
sssp.get_path(vid='Weyerhaeuser', show=True,
              highlight=['Microsoft', 'Weyerhaeuser'], arrows=True, ewidth=1.5)
```

```
[('Microsoft', 0.0),
 ('Google', 1.0),
 ('Tax avoidance', 2.0),
 ('Weyerhaeuser', 3.0)]
```



Clustering

Clustering is the task of grouping data so that points in the same cluster are highly similar to each other, while points in different clusters are dissimilar. Clustering is a form of unsupervised learning because there is no target variable indicating which groups the training data belong to.

The [GraphLab clustering toolkit](#) includes two models: K-Means and DBSCAN. **K-Means** finds cluster centers for a predetermined number of clusters ("K") by minimizing the sum of squared distances from each point to its assigned cluster. Points are assigned to the cluster whose center is closest. It is usually the faster of the two options, and can be accelerated further by setting the `batch_size` parameter to use only a small subset of data for each training iteration.

DBSCAN is the most popular probability density-based clustering method. It creates clusters by connecting neighboring points that have high estimated probability density. Although less computationally efficient than K-Means, DBSCAN captures more flexible cluster shapes, automatically determines the best number of clusters, and detects outliers.

K-means

K-means finds cluster centers for a predetermined number of clusters ("K") by minimizing the sum of squared distances from each point to its assigned cluster. Points are assigned to the cluster whose center is closest.

Lloyd's algorithm is the standard way to compute K-means clusters, and it describes the essential intuition for the method. After initial centers are chosen, two steps repeat until the cluster assignment no longer changes for any point (which is equivalent to the cluster centers no longer moving):

1. Assign each point to the cluster with the closest center.
2. Update each cluster center to the be mean of the assigned points.

The GraphLab Create implementation of K-means uses several wrinkles to improve the speed of the method and quality of the results. Initial cluster centers are chosen with the K-means++ algorithm (if not provided explicitly by the user). This algorithm chooses cluster centers that are far apart with high probability, which tends to reduce the number of iterations needed for convergence, and make it less likely that the method returns sub-optimal results.

In addition, GraphLab Create's K-means uses the triangle inequality to reduce the number of exact distances that need to be computed in each iteration. Conceptually, if we know that a data point x is close to center A , which is in turn far from center B , then there is no need to compute the exact distance from point x to center B when assigning x to a cluster.

Basic Usage

We illustrate usage of GraphLab Create K-means with the dataset from the [June 2014 Kaggle competition to classify schizophrenic subjects based on MRI scans](#). The original data consists of two sets of features: functional network connectivity (FNC) features and source-based morphometry (SBM) features, which we incorporate into a single `SFrame` with `SFrame.join`. For convenience the data can be downloaded from our public AWS S3 bucket; the following code snippet does this if the data is not found in the local working directory.

```

import os
import graphlab as gl

if os.path.exists('schizophrenia_clean'):
    sf = gl.SFrame('schizophrenia_clean')
else:
    sf_functional = gl.SFrame.read_csv(
        'https://static.turi.com/datasets/mlsp_2014/train_FNC.csv')
    sf_morphometry = gl.SFrame.read_csv(
        'https://static.turi.com/datasets/mlsp_2014/train_SBM.csv')

    sf = sf_functional.join(sf_morphometry, on="Id")
    sf = sf.remove_column('Id')

sf.save('schizophrenia_clean')

```

The most basic usage of K-means clustering requires only a choice for the number of clusters, K . We rarely know the correct number of clusters *a priori*, but the following simple heuristic sometimes works well:

$$K \approx \sqrt{n/2}$$

where n is the number of rows in your dataset. By default, the maximum number of iterations is 10, and all features in the input dataset are used.

Analogous to all other GraphLab Create toolkits the model is created through the `kmeans.create` API:

```

from math import sqrt
K = int(sqrt(sf.num_rows() / 2.0))

kmeans_model = gl.kmeans.create(sf, num_clusters=K)
kmeans_model.summary()

```

```

Class : KmeansModel

Schema
-----
Number of clusters      : 6
Number of examples       : 86
Number of feature columns: 410
Number of unpacked features: 410
Row label name          : row_id

Training Summary
-----
Training method          : elkan
Number of training iterations : 2
Batch size                : 86
Total training time (seconds) : 0.2836

Accessible fields :
  cluster_id            : An SFrame containing the cluster assignments.
  cluster_info           : An SFrame containing the cluster centers.

```

The model summary shows the usual fields about model schema, training time, and training iterations. It also shows that the K-means results are returned in two SFrames contained in the model: `cluster_id` and `cluster_info`. The `cluster_info` SFrame indicates the final cluster centers, one per row, in terms of the same features used to create the model.

```
kmeans_model['cluster_info'].print_rows(num_columns=5, max_row_width=80,
                                         max_column_width=10)
```

```
+-----+-----+-----+-----+-----+
|   FNC1 |   FNC2 |   FNC3 |   FNC4 |   FNC5 | ... |
+-----+-----+-----+-----+-----+
| 0.1870... | 0.0801... | -0.092... | -0.0957298 | 0.0893... | ... |
| 0.21752 | 0.0363... | -0.027... | -0.063... | 0.0556... | ... |
| 0.2293... | 0.1017... | -0.046... | -0.051... | 0.2313... | ... |
| 0.1654... | -0.156... | -0.327... | -0.278... | -0.033... | ... |
| 0.2549... | 0.02532 | 0.0081... | -0.134... | 0.3875... | ... |
| 0.1072... | 0.0754... | -0.119422 | -0.312... | 0.1100... | ... |
+-----+-----+-----+-----+-----+
[6 rows x 413 columns]
```

The last three columns of the `cluster_info` SFrame indicate metadata about the corresponding cluster: ID number, number of points in the cluster, and the within-cluster sum of squared distances to the center.

```
kmeans_model['cluster_info'][['cluster_id', 'size', 'sum_squared_distance']]
```

```
+-----+-----+-----+
| cluster_id | size | sum_squared_distance |
+-----+-----+-----+
|     0      |   7   |    340.44890213   |
|     1      |  11   |    533.886421204  |
|     2      |  49   |   2713.56332016   |
|     3      |  13   |    714.04801178   |
|     4      |   3   |   177.421077728   |
|     5      |   3   |   151.59986496   |
+-----+-----+-----+
[6 rows x 3 columns]
```

The `cluster_id` field of the model shows the cluster assignment for each input data point, along with the Euclidean distance from the point to its assigned cluster's center.

```
kmeans_model['cluster_id'].head()
```

```
+-----+-----+-----+
| row_id | cluster_id |      distance   |
+-----+-----+-----+
|     0   |       3   |  6.52821207047 |
|     1   |       2   |  6.45124673843 |
|     2   |       2   |  7.58535766602 |
|     3   |       2   |  7.64395523071 |
|     4   |       3   |  7.42247104645 |
|     5   |       2   |  8.29837036133 |
|     6   |       4   |  7.61347103119 |
|     7   |       2   |  6.98522281647 |
|     8   |       2   |  8.56831073761 |
|     9   |       0   |  7.91477823257 |
+-----+-----+-----+
[10 rows x 3 columns]
```

Assigning New Points to Clusters

New data points can be assigned to the clusters of a K-means model with the `KmeansModel.predict` method. For K-means, the assignment is simply the nearest cluster center (in Euclidean distance), which is how the training data are assigned as well. Note that the model's cluster centers are *not updated* by the `predict` method.

For illustration purposes, we predict the cluster assignments for the first 5 rows of our existing data. The assigned clusters are identical to the assignments in the model results (above), which is a good sanity check.

```
new_clusters = kmeans_model.predict(sf[:5])
new_clusters
```

```
dtype: int
Rows: 5
[3, 2, 2, 2, 3]
```

Advanced Usage

For large datasets K-means clustering can be a time-consuming method. One simple way to reduce the computation time is to reduce the number of training iterations with the `max_iterations` parameter. The model prints a warning during training to indicate that the algorithm stops before convergence is reached.

```
kmeans_model = gl.kmeans.create(sf, num_clusters=K, max_iterations=1)
```

```
PROGRESS: WARNING: Clustering did not converge within max_iterations.
```

It can also save time to set the initial centers manually, rather than having the tool choose the initial centers automatically. These initial centers can be chosen randomly from a sample of the original dataset, then passed to the final K-means model.

```
kmeans_sample = gl.kmeans.create(sf.sample(0.2), num_clusters=K,
                                  max_iterations=0)

my_centers = kmeans_sample['cluster_info']
my_centers = my_centers.remove_columns(['cluster_id', 'size',
                                         'sum_squared_distance'])

kmeans_model = gl.kmeans.create(sf, initial_centers=my_centers)
```

For really large datasets, the tips above may not be enough to get results in a reasonable amount of time; in this case, we can switch to **minibatch K-means**, using the same GraphLab Create model. The `batch_size` parameter indicates how many randomly sampled points to use in each training iteration when updating cluster centers. Somewhat counter-intuitively, the results for minibatch K-means tend to be very similar to the exact algorithm, despite typically using only a small fraction of the training data in each iteration. Note that for the minibatch version of K-means, the model will always compute a number of iterations equal to `max_iterations`; it does not stop early.

```
kmeans_model = gl.kmeans.create(sf, num_clusters=K, batch_size=30,
                                 max_iterations=10)
kmeans_model.summary()
```

```
Class : KmeansModel

Schema
-----
Number of clusters      : 6
Number of examples       : 86
Number of feature columns : 410
Number of unpacked features : 410
Row label name          : row_id

Training Summary
-----
Training method          : minibatch
Number of training iterations : 10
Batch size                : 30
Total training time (seconds) : 0.3387

Accessible fields :
  cluster_id            : An SFrame containing the cluster assignments.
  cluster_info           : An SFrame containing the cluster centers.
```

The model summary shows the training method here is "minibatch" with our specified batch size of 30, unlike the previous model which used the "elkan" (exact) method with a batch size of 86 - the total number of examples in our dataset.

References and more information

- [Wikipedia - k-means clustering](#)
- Artuhur, D. and Vassilvitskii, S. (2007) [k-means++: The Advantages of Careful Seeding](#). In Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 1027-1035.
- Elkan, C. (2003) [Using the triangle inequality to accelerate k-means] (<http://www.aaai.org/Papers/ICML/2003/ICML03-022.pdf>). In Proceedings of the Twentieth International Conference on Machine Learning, Volume 3, pp. 147-153.
- Sculley, D. (2010) [Web Scale K-Means Clustering](#). In Proceedings of the 19th International Conference on World Wide Web. pp. 1177-1178

DBSCAN

DSBCAN, short for Density-Based Spatial Clustering of Applications with Noise, is the most popular density-based clustering method. Density-based clustering algorithms attempt to capture our intuition that a cluster — a difficult term to define precisely — is a region of the data space where there are lots of points, surrounded by a region where there are few points. DBSCAN does this by partitioning the input data points into three types:

- *Core* points have a large number of other points within a given neighborhood. The parameter `min_core_neighbors` defines how many points counts as a "large number", while the `radius` parameter defines how large the neighborhoods are around each point. Specifically, a point y is in the neighborhood of point x if $d(x, y) < \text{radius}$, where d is a user-specified distance function.
- *Boundary* points are within distance `radius` of a core point, but don't have sufficient neighbors of their own to be considered core.
- *Noise* points comprise the remainder of the data. These points have too few neighbors to be considered core points, and are further than distance `radius` from all core points.

Clusters are formed by connecting core points that are neighbors of each other, then assigning boundary points to their nearest core neighbor's cluster. Noise points are left unassigned.

DBSCAN tends to be slower than K-means because it requires computation of the similarity graph on the input dataset, but it has several major conceptual advantages:

- The number of clusters does not need to be known *a priori*; DBSCAN detects the optimum number of clusters automatically for the given values of `min_core_neighbors` and `radius`.
- DBSCAN recovers much more flexible cluster shapes than K-means, which can only find spherical clusters.
- DBSCAN intrinsically finds and labels outliers as such, making it a great tool for outlier and anomaly detection.
- DBSCAN works with any distance function. Note that results may be poor for distances that do not obey standard properties of distances, i.e. symmetry, non-negativity, triangle inequality, and identity of indiscernibles. The distances "euclidean", "manhattan", "jaccard", and "levenshtein" will likely yield the best results.

Basic usage

To illustrate the basic usage of DBSCAN and how the results can differ from K-means, we simulate non-spherical, low-dimensional data using the scikit-learn datasets module.

```
import graphlab as gl
from sklearn.datasets import make_moons

data = make_moons(n_samples=200, shuffle=True, noise=0.1, random_state=19)
sf = gl.SFrame(data[0]).unpack('X1')

dbscan_model = gl.dbscan.create(sf, radius=0.25)
dbscan_model.summary()
```

```
Class : DBSCANModel

Schema
-----
Number of examples      : 200
Number of feature columns : 2
Max distance to a neighbor (radius) : 0.25
Min number of neighbors for core points : 10
Number of distance components : 1

Training summary
-----
Total training time (seconds)      : 0.1954
Number of clusters                 : 2

Accessible fields
-----
cluster_id : Cluster label for each row in the input dataset
```

Like the K-means model, the assignments of points to clusters is in the models' `cluster_id` field. The second column shows the cluster assignment for the row index of the input data indicated by the first column. The third column shows whether DBSCAN considers the point *core*, *boundary*, or *noise*. Noise points are not assigned to a cluster, which is encoded as a "missing" value in the 'cluster_id' column.

```
dbscan_model['cluster_id'].head(5)
```

```
+-----+-----+-----+
| row_id | cluster_id | type |
+-----+-----+-----+
| 175    |      1     | core |
| 136    |      0     | core |
| 33     |      1     | core |
| 113    |      0     | core |
| 110    |      1     | core |
+-----+-----+-----+
[5 rows x 3 columns]
```

```
dbscan_model['cluster_id'].tail(5)
```

```
+-----+-----+-----+
| row_id | cluster_id | type   |
+-----+-----+-----+
| 53    |      0     | boundary |
| 4     |      1     | boundary |
| 43    |      0     | boundary |
| 116   |      0     | boundary |
| 74    |    None    | noise   |
+-----+-----+-----+
[5 rows x 3 columns]
```

Because we generated 2D data, we can plot it and color the points according to the cluster assignments generated by our DBSCAN model. The first step is to join the cluster results back to the original data. Please note: DBSCAN scrambles the row order - be careful!

Next we define boolean masks so we can plot the core, boundary, and noise points separately. Boundary points are drawn smaller than core points, and (unclustered) noise points are left black. Some plotting code is omitted for brevity.

```

import matplotlib.pyplot as plt
plt.style.use('ggplot')

sf = sf.add_row_number('row_id')
sf = sf.join(dbSCAN_model['cluster_id'], on='row_id', how='left')
sf = sf.rename({'cluster_id': 'dbSCAN_id'})

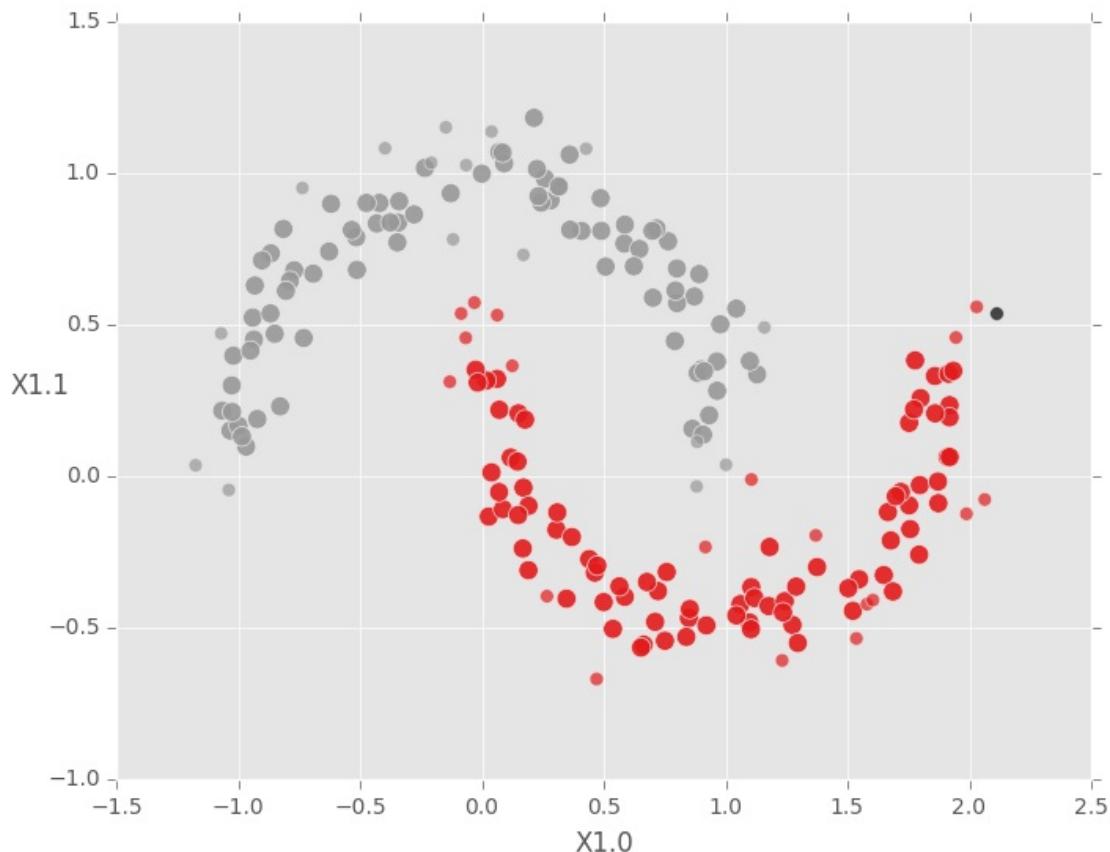
core_mask = sf['type'] == 'core'
boundary_mask = sf['type'] == 'boundary'
noise_mask = sf['type'] == 'noise'

fig, ax = plt.subplots()
ax.scatter(sf['X1.0'][core_mask], sf['X1.1'][core_mask], s=80, alpha=0.9,
           c=sf['dbSCAN_id'][core_mask], cmap=plt.cm.Set1)

ax.scatter(sf['X1.0'][boundary_mask], sf['X1.1'][boundary_mask], s=40,
           alpha=0.7, c=sf['dbSCAN_id'][boundary_mask], cmap=plt.cm.Set1)

ax.scatter(sf['X1.0'][noise_mask], sf['X1.1'][noise_mask], s=40, alpha=0.7,
           c='black')
fig.show()

```



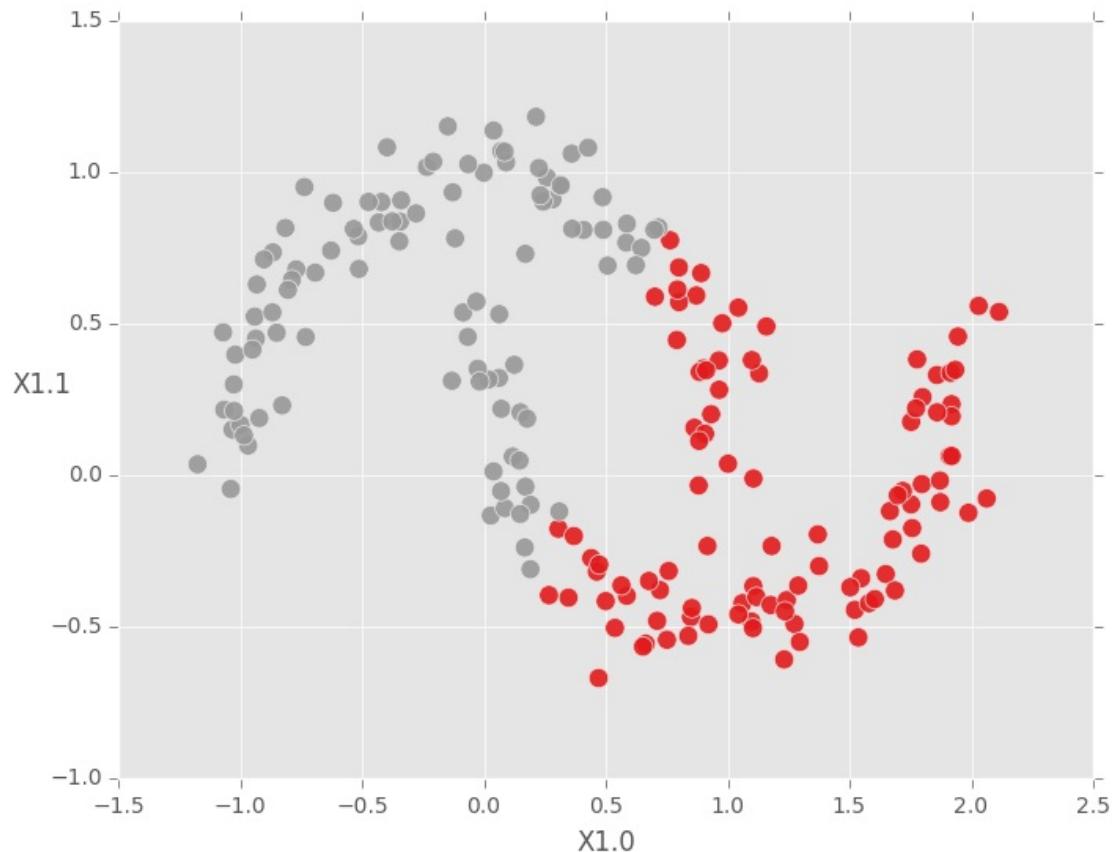
For comparison, K-means cannot identify the true clusters in this case, even when we tell the model the correct number of clusters.

```

kmeans_model = gl.kmeans.create(sf, features=['X1.0', 'X1.1'], num_clusters=2)
sf['kmeans_id'] = kmeans_model['cluster_id']['cluster_id']

fig, ax = plt.subplots()
ax.scatter(sf['X1.0'], sf['X1.1'], s=80, alpha=0.9, c=sf['kmeans_id'],
           cmap=plt.cm.Set1)
fig.show()

```



Setting key parameters

DBSCAN is particularly useful when the number of clusters is not known *a priori*, but it can be tricky to set the `radius` and `min_core_neighbors` parameters. To improve the quality of DBSCAN results, try the following:

- When `radius` is too large or `min_core_neighbors` is too small, every point being labeled as a core point, often connected as a single cluster. If your trained model has a single cluster with only core points (and you suspect this is incorrect), try decreasing the `radius` parameter or increasing the `min_core_neighbors` parameter.
- Conversely, if DBSCAN labels all points as noise, with no clusters returned at all, try increasing the `radius` or decreasing the `min_core_neighbors`.

- Use the [GraphLab Create nearest neighbors toolkit](#) to construct a similarity graph on the data and plot the distribution of distances with Canvas. This will give you a sense for reasonable values of the `radius` parameter, then the `min_core_neighbors` parameter can be tuned by itself for optimal results.

Choosing the distance function

DBSCAN is not restricted to Euclidean distances. The GraphLab Create implementation allows any distance function---including composite distances---to be used with DBSCAN. This allows for a tremendous amount of flexibility in terms of data types and tuning for optimal clustering quality. To demonstrate this flexibility we'll use DBSCAN with Jaccard distance to deduplicate wikipedia articles.

```
import graphlab as gl
import os
if os.path.exists('wikipedia_w16'):
    sf = gl.SFrame('wikipedia_w16')
else:
    sf = gl.SFrame.read_csv('https://static.turi.com/datasets/wikipedia/raw/w16.csv',
                           header=False)
sf.save('wikipedia_w16')
```

This particular subset of wikipedia has over 72,000 documents; in the interest of speed for the demo we sample 20% of this. We also preprocess the data by constructing a bag-of-words representation for each article and trimming out stopwords. See GraphLab Create's [text analytics](#) and [SArray](#) documentation for more details.

```
sf_sample = sf.sample(0.2)
sf_sample['word_bag'] = gl.text_analytics.count_words(sf_sample['X1'])
sf_sample['word_bag'] = sf_sample['word_bag'].dict_trim_by_keys(
    gl.text_analytics.stopwords(), exclude=True)
```

With our trimmed bag of words representation, Jaccard distance is a natural choice. For the purpose of deduplication we want to identify points as "core" if they have any near neighbors at all, so we set `min_core_neighbors` to 1. To define what we mean by "near" we set the `radius` parameter somewhat arbitrarily at 0.5; this means that two points are considered neighbors if they share 50% or more of the words present in either article.

```
wiki_cluster = gl.dbscan.create(sf_sample, features=['word_bag'],
                                 distance='jaccard', radius=0.5,
                                 min_core_neighbors=1)
wiki_cluster.summary()
```

```
Class : DBSCANModel

Schema
-----
Number of examples : 14265
Number of feature columns : 1
Max distance to a neighbor (radius) : 0.5
Min number of neighbors for core points : 1
Number of distance components : 1

Training summary
-----
Total training time (seconds) : 35.1235
Number of clusters : 88

Accessible fields
-----
cluster_id : Cluster label for each row in the input dataset
```

From the model summary we see there are 88 clusters in the set of 14,265 documents. For more detail on the distribution of cluster sizes, we can use the method.

```
wiki_cluster['cluster_id']['cluster_id'].sketch_summary()
```

item	value	is exact
Length	14265	Yes
Min	0.0	Yes
Max	87.0	Yes
Mean	32.3243243243	Yes
Sum	10764.0	Yes
Variance	699.186105024	Yes
Standard Deviation	26.4421274678	Yes
# Missing Values	13932	Yes
# unique values	88	No

Most frequent items:

value	1	10	5	19	9	36	64	62	11	23
count	33	29	18	15	14	13	9	7	7	5

Quantiles:

0%	1%	5%	25%	50%	75%	95%	99%	100%
0.0	1.0	1.0	10.0	24.0	55.0	80.0	86.0	87.0

This indicates that of our 14,265 documents, 13,932 are considered noise (i.e. missing values), which in this context means they have no duplicates. The largest cluster has 33 duplicate documents! Let's see what they are. To do this we again need to join the cluster IDs back to our input dataset.

```
sf_sample = sf_sample.add_row_number('row_id')
sf_sample = sf_sample.join(wiki_cluster['cluster_id'], on='row_id', how='left')
```

```
In [98]: sf_sample[sf_sample['cluster_id'] == 1][['X1']].print_rows(10, max_row_width=80,
+-----+
| X1
+-----+
| graceunitedmethodistchurchwilmingtondelaware it was built in 1868 and added ...
| firstpresbyterianchurchdelhinewyork it was added to the national register of...
| methodistepiscopalchurchofnorwich it was added to the national register of h...
| odonelhouseandfarm it was listed on the national register of historic places...
| saintpaulsepiscopalchurchwatertownnewyork it was listed on the national regi...
| windsorhillshistoricdistrict it was added to the national register of histor...
| pepperellcenterhistoricdistrict the district was added to the national regis...
| eboardmanhouse it was built in 1820 and added to the national register of hi...
| johnsoutherhouse the house was built in 1883 and added to the national regis...
| ednastolikerthreeedecker it was built in 1916 and added to the national regis...
+-----+
[33 rows x 1 columns]
```

It seems this cluster captures a set of article stubs that simply list when a physical structure was build and added to the National Register of Historic Places.

There are two important caveats regarding distance functions in DBSCAN:

1. DBSCAN computes many pairwise distances. For dense data, GraphLab Create computes some distances much faster than others, namely "euclidean", "squared_euclidean", "cosine", and "transformed_dot_product". Other distances, as well as all distances with sparse data, may result in longer run times.
2. DBSCAN does not explicitly require the standard distance properties (symmetry, non-negativity, triangle inequality, and identity of indiscernibles) to hold, but it is based on connecting high-density points which are *close* to each other into a single cluster. If the specified notion of closeness violates the usual distance properties, DBSCAN may yield counterintuitive results. We expect to see the most intuitive results with "euclidean", "manhattan", "jaccard", and "levenshtein" distances.

References and more information

- Ester, M., et al. (1996) [A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise](#). In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining. pp. 226-231.
- [Wikipedia - DBSCAN](#)
- [Visualizing DBSCAN Clustering](#)

Nearest Neighbors

The GraphLab Create [nearest neighbors toolkit](#) is used to find the rows in a data table that are most similar to a query row. This is a two-stage process, analogous to many other GraphLab Create toolkits. First we create a [NearestNeighborsModel](#), using a [reference dataset](#) contained in an [SFrame](#). Next we query the model, using either the `query` or the `similarity_graph` method. Each of these methods is explained further below.

For this chapter we use an example dataset of house attributes and prices, downloaded from Turi's public datasets bucket.

```
import graphlab as gl
import os

if os.path.exists('houses.csv'):
    sf = gl.SFrame.read_csv('houses.csv')
else:
    data_url = 'https://static.turi.com/datasets/regression/houses.csv'
    sf = gl.SFrame.read_csv(data_url)
    sf.save('houses.csv')

sf.head(5)
```

tax	bedroom	bath	price	size	lot
590	2	1.0	50000	770	22100
1050	3	2.0	85000	1410	12000
20	3	1.0	22500	1060	3500
870	2	2.0	90000	1300	17500
1320	3	2.0	133000	1500	30000

[5 rows x 6 columns]

Because the features in this dataset have very different scales (e.g. price is in the hundreds of thousands while the number of bedrooms is in the single digits), it is important to **normalize the features**. In this example we standardize so that each feature is measured in terms of standard deviations from the mean (see [Wikipedia for more detail](#)). In addition, both reference and query datasets may have a column with row labels, but for this example we let the model default to using row indices as labels.

```
for c in sf.column_names():
    sf[c] = (sf[c] - sf[c].mean()) / sf[c].std()
```

First, we **create a nearest neighbors model**. We can list specific features to use in our distance computations, or default to using all features in the reference SFrame. In the model summary below the following code snippet, note that there are three features, because our second command specifies three numeric SFrame columns as features for the model. There are also three unpacked features, because each feature is in its own column.

```
model = gl.nearest_neighbors.create(sf)
model = gl.nearest_neighbors.create(sf, features=['bedroom', 'bath', 'size'])
model.summary()
```

Class : NearestNeighborsModel

Attributes

 Distance : euclidean
 Method : ball tree
 Number of examples : 15
 Number of feature columns : 3
 Number of unpacked features : 3
 Total training time (seconds) : 0.0091

Ball Tree Attributes

 Tree depth : 1
 Leaf size : 1000

To retrieve the five closest neighbors for *new* data points or a *subset* of the original reference data, we **query** the model with the `query` method. Query points must also be contained in an SFrame, and must have columns with the same names as those used to construct the model (additional columns are allowed, but ignored). The result of the `query` method is an SFrame with four columns: query label, reference label, distance, and rank of the reference point among the query point's nearest neighbors.

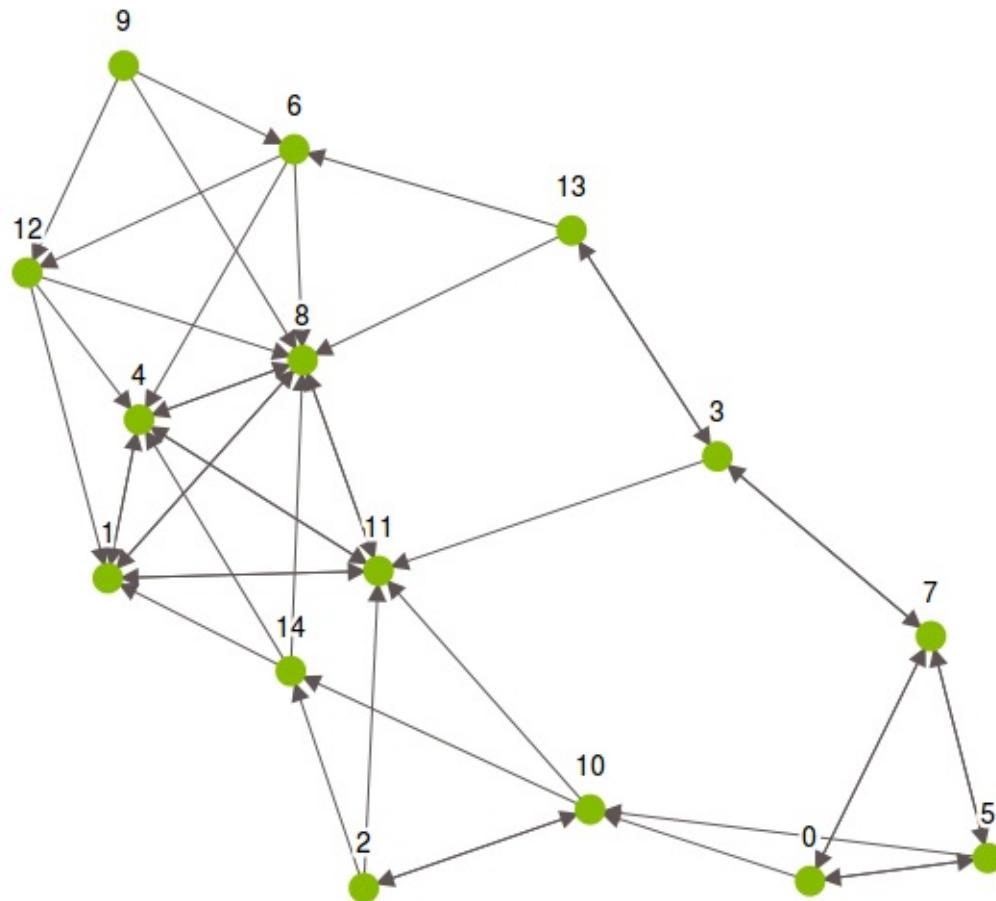
```
knn = model.query(sf[:5], k=5)
knn.head()
```

query_label	reference_label	distance	rank
0	0	0.0	1
0	5	0.100742954001	2
0	7	0.805943632008	3
0	10	1.82070683014	4
0	2	1.83900997922	5
1	1	0.0	1
1	8	0.181337317202	2
1	4	0.181337317202	3
1	11	0.322377452803	4
1	12	0.705200678007	5

[10 rows x 4 columns]

In some cases the query dataset *is* the reference dataset. For this task of constructing the **similarity_graph** on the reference data, the model's `similarity_graph` can be used. For brute force models it can be almost twice as fast, depending on the data sparsity and chosen distance function. By default, the `similarity_graph` method returns an **SGraph** whose vertices are the rows of the reference dataset and whose edges indicate a nearest neighbor match. Specifically, the destination vertex of an edge is a nearest neighbor of the source vertex. `similarity_graph` can also return results in the same form as the `query` method if so desired.

```
sim_graph = model.similarity_graph(k=3)
sim_graph.show(vlabel='id', arrows=True)
```



Distance functions

The most critical choice in computing nearest neighbors is the **distance function** that measures the dissimilarity between any pair of observations.

For numeric data, the options are `euclidean`, `manhattan`, `cosine`, and `transformed_dot_product`. For data in dictionary format (i.e. sparse data), `jaccard` and `weighted_jaccard` are also options, in addition to the numeric distances. For string features, use `levenshtein` distance, or use the text analytics toolkit's `count_ngrams` feature to convert strings to dictionaries of words or character shingles, then use Jaccard or weighted Jaccard distance. Leaving the distance parameter set to its default value of `auto` tells the model to choose the most reasonable distance based on the type of features in the reference data. In the following output cell, the second line of the model summary confirms our choice of Manhattan distance.

```
model = gl.nearest_neighbors.create(sf, features=['bedroom', 'bath', 'size'],
                                     distance='manhattan')
model.summary()
```

```

Class : NearestNeighborsModel

Attributes
-----
Distance : manhattan
Method : ball tree
Number of examples : 15
Number of feature columns : 3
Number of unpacked features : 3
Total training time (seconds) : 0.013

Ball Tree Attributes
-----
Tree depth : 1
Leaf size : 1000

```

Distance functions are also exposed in the `graphlab.distances` module. This allows us not only to specify the distance argument for a nearest neighbors model as a distance function (rather than a string), but also to *use* that function for any other purpose.

In the following snippet we use a nearest neighbors model to find the closest reference points to the first three rows of our dataset, then confirm the results by computing a couple of the distances manually with the Manhattan distance function.

```

model = gl.nearest_neighbors.create(sf, features=['bedroom', 'bath', 'size'],
                                    distance=gl.distances.manhattan)
knn = model.query(sf[:3], k=3)
knn.print_rows()

sf_check = sf[['bedroom', 'bath', 'size']]
print "distance check 1:", gl.distances.manhattan(sf_check[2], sf_check[10])
print "distance check 2:", gl.distances.manhattan(sf_check[2], sf_check[14])

```

```
+-----+-----+-----+
| query_label | reference_label |      distance | rank |
+-----+-----+-----+
|     0       |        0       |      0.0      |  1   |
|     0       |        5       |  0.100742954001 |  2   |
|     0       |        7       |  0.805943632008 |  3   |
|     1       |        1       |      0.0      |  1   |
|     1       |        8       |  0.181337317202 |  2   |
|     1       |        4       |  0.181337317202 |  3   |
|     2       |        2       |      0.0      |  1   |
|     2       |       10      |  0.0604457724006 |  2   |
|     2       |       14      |  1.61656820464  |  3   |
+-----+-----+-----+
[9 rows x 4 columns

distance check 1: 0.0604457724006
distance check 2: 1.61656820464
```

GraphLab Create also allows **composite distances**, which allow the nearest neighbors tool (and other distance-based tools) to work with features that have different types. Specifically, a composite distance is a *weighted sum of standard distances applied to subsets of features*, specified in the form of a Python list. Each element of a composite distance list contains three things:

1. a list or tuple of feature names
2. a standard distance name
3. a multiplier for the standard distance.

In our house price dataset, for example, suppose we want to measure the difference between numbers of bedrooms and baths with Manhattan distance and the difference between house and lot sizes with Euclidean distance. In addition, we want the Euclidean component to carry twice as much weight. The composite distance for this would be:

```
my_dist = [[[ 'bedroom', 'bath'], 'manhattan', 1],
           [['size', 'lot'], 'euclidean', 2]]
```

This list can be passed to the `distance` parameter just like a standard distance function name or handle.

```
model = gl.nearest_neighbors.create(sf, distance=my_dist)
model.summary()
```

```

Class : NearestNeighborsModel

Attributes
-----
Method : brute_force
Number of distance components : 2
Number of examples : 15
Number of feature columns : 4
Number of unpacked features : 4
Total training time (seconds) : 0.0017

```

If we specify the distance parameter as `auto`, a composite distance is created where each type of feature is paired with the most appropriate distance function. Please see the documentation for the [GraphLab Create distances module](#) for more on composite distances.

Search methods

Another important choice in model creation is the **method**. The `brute_force` method computes the distance between a query point and *each* of the reference points, with a run time linear in the number of reference points. Creating a model with the `ball_tree` method takes more time, but leads to much faster queries by partitioning the reference data into successively smaller balls and searching only those that are relatively close to the query. The default method is `auto` which chooses a reasonable method based on both the feature types and the selected distance function. The method parameter is also specified when the model is created. The third row of the model summary confirms our choice to use the ball tree in the next example.

```

model = gl.nearest_neighbors.create(sf, features=['bedroom', 'bath', 'size'],
                                     method='ball_tree', leaf_size=5)
model.summary()

```

```
Class : NearestNeighborsModel

Attributes
-----
Method : ball_tree
Number of distance components : 1
Number of examples : 15
Number of feature columns : 3
Number of unpacked features : 3
Total training time (seconds) : 0.0253

Ball Tree Attributes
-----
Tree depth : 3
Leaf size : 5
```

If the ball tree is used, it's important to choose an appropriate value for the 'leaf_size' parameter, which controls how many observations are stored in each leaf of the ball tree. By default, this is set so that the tree is no more than 12 levels deep, but larger or smaller values may lead to quicker queries depending on the shape and dimension of the data. Our houses example only has 15 rows, so the `leaf_size` parameter (and the `ball_tree` method for that matter) are not too useful, but for illustration purposes we set the leaf size to 5 above.

Missing data

The reference dataset that is used to create the nearest neighbors model cannot have missing data. Please use the `SFrame.fillna` and `SFrame.dropna` utilities to preprocess missing values before creating a nearest neighbors model.

On the other hand, data passed to the `query` method *can* have missing data. For numeric columns, missing values are imputed to be the mean of the corresponding column in the reference dataset used to create the model. Missing values in string columns are imputed as empty strings.

Text analysis

In this chapter we will show how to do standard text analytics using GraphLab Create.

Applications for automated text analytics include:

- detecting user sentiment regarding product reviews
- creating features for use in other machine learning models
- understanding large collections of documents

Analysis

Suppose our text data is currently arranged into a single file, where each line of that file contains all of the text in a single document. Here we can use `SFrame.read_csv` to parse the text data into a one-column SFrame.

```
import os
if os.path.exists('wikipedia_w16'):
    sf = graphlab.SFrame('wikipedia_w16')
else:
    sf = graphlab.SFrame.read_csv('https://static.turi.com/datasets/wikipedia/raw/w16.csv')
    sf.save('wikipedia_w16')
```

sf

Columns:

X1 str

Rows: 72269

Data:

X1
alainconnes alain connes i ...
americannationalstandardsi ...
alberteinstein near the be ...
austriangerman as german i ...
arsenic arsenic is a metal ...
alps the alps alpen alpi a ...
alexiscarrel born in saint ...
adelaide adelaide is a coa ...
artist an artist is a pers ...
abdominalsurgery the three ...
...

[72269 rows x 1 columns]

Note: Only the head of the SFrame is printed.

You can use `print_rows(num_rows=m, num_columns=n)` to print more rows and columns.

Bag-of-words

Both SFrames and SArrays expose functionality that can be very useful for manipulating text data. For example, one common preprocessing task for text data is to transform it into "bag-of-words" format: each document is represented by a map where the words are keys and the values are the number of occurrences. So a document containing the text "hello goodbye hello" would be represented by a `dict` type element containing the value `{"hello": 2, "goodbye":1}`. This transformation can be accomplished with the following code.

```
bow = graphlab.text_analytics.count_words(sf['x1'])
```

We can print five of the words in the first document

```
bow[0].keys()[:5]
```

```
['and', 'work', 'baumconnes', 'gold', 'almost']
```

and find the documents that contain the word "gold":

```
bow.dict_has_any_keys(['gold'])
```

We can save this representation of the documents as another column of the original SFrame.

```
sf['bow'] = bow
```

TF-IDF

Another useful representation for text data is called TF-IDF (term frequency - inverse document frequency). This is a modification of the bag-of-words format where the counts are transformed into scores: words that are common across the document corpus are given low scores, and rare words occurring often in a document are given high scores.

$$\text{TF-IDF}(word, document) = N(word, document) * \log(1 / \sum_d N(word, d))$$

where $N(w, d)$ is the number of times word w occurs in document d. This transformation can be done to an SArray of dict type containing documents in bow-of-words format using [tf_idf](#).

```
sf['tfidf'] = graphlab.text_analytics.tf_idf(sf['bow'])
```

BM25

The BM25 score is yet another useful representation for text data. It scores each document in a corpus according to the document's relevance to a particular query. For a query with terms q_1, \dots, q_n , the BM25 score for document d is:

$$\text{BM25}(d) = \sum_{i=1}^n \text{IDF}(q_i) \frac{f(q_i) * (k_1 + 1)}{f(q_i) + k_1 * (1 - b + b * |D|/d_{avg})}$$

where:

- $f(q_i)$ is the number of times term q_i occurs in document d ,
- $|D|$ is the number of words in document d ,
- d_{avg} is the average number of words per document,
- b and k_1 are free parameters for Okapi BM25,

The first quantity in the sum is the inverse document frequency. For a corpus with N documents, inverse document frequency for term q_i is:

$$\text{IDF}(q_i) = \log \frac{N - N(q_i) + 0.5}{N(q_i) + 0.5}$$

where $N(q_i)$ is the number of documents in the corpus that contain term q_i .

The transformed output is a column of type float with the BM25 score for each document. For more details on the BM25 score see http://en.wikipedia.org/wiki/Okapi_BM25.

```
query = ['beatles', 'john', 'paul']
bm25_scores = graphlab.text_analytics.bm25(dataset, query)
```

Text cleaning

We can easily remove all words do not occur at least twice in each document using `SArray.dict_trim_by_values`.

```
docs = sf['bow'].dict_trim_by_values(2)
```

Alternatively, we can remove all words which do not occur at least `threshold` number of times using the `RareWordTrimmer`.

GraphLab Create also contains a helper function called `stopwords` that returns a list of common words. We can use `SArray.docs.dict_trim_by_keys` to remove these words from the documents as a preprocessing step. NB: Currently only English words are available.

```
docs = docs.dict_trim_by_keys(graphlab.text_analytics.stopwords(), exclude=True)
```

To confirm that we have indeed removed common words, e.g. "and", "the", etc, we can examine the first document.

```
docs[0]
```

```
{'academy': 5,
'algebras': 2,
'connes': 3,
'differential': 2,
'early': 2,
'geometry': 2,
'including': 2,
'medal': 2,
'operator': 2,
'physics': 2,
'sciences': 5,
'theory': 2,
'work': 2}
```

Tokenization

For an SArray of strings, where each row is assumed to be a natural English language document, the tokenizer transforms each row into an ordered list of strings that represents the a simpler version of the Penn-Tree-Bank-style (PTB-style) tokenization of that row's document. For many text analytics tasks that require word-level granularity, simple space delimitation does not address some of the subtleties of natural language text, especially with respect to contractions, sentence-final punctuation, URL's, email addresses, phone numbers, and other quirks. The representation of a document provided by PTB-style of tokenization is essential for sequence-tagging, parsing, bag-of-words treatment, and any text analytics task that requires word-level granularity. For a description of this style of tokenization, see <https://www.cis.upenn.edu/~treebank/tokenization.html>.

```
tokenized_docs = graphlab.SFrame()
tokenized_docs['tokens'] = graphlab.text_analytics.tokenize(sf['X1'])
tokenized_docs
```

```

Columns:
tokens  list

Rows: 72269

Data:
+-----+
|      tokens      |
+-----+
| [alainconnes, alain, conne... | 
| [americannationalstandards... | 
| [alberteinstein, near, the... | 
| [austriangerman, as, germa... | 
| [arsenic, arsenic, is, a, ... | 
| [alps, the, alps, alpen, a... | 
| [alexiscarrel, born, in, s... | 
| [adelaide, adelaide, is, a... | 
| [artist, an, artist, is, a... | 
| [abdominalsurgery, the, th... | 
+-----+
[72269 rows x 1 columns]
Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.

```

Note that our tokenizer does not normalize quote and bracket-like characters as described by the linked document.

Part of Speech Extraction

It can be useful to extract particular parts of speech. Specifically, you may want to highlight unique nouns in your text, identify adjectives with the high sentiment scores, or pull out nouns to generate candidate entities. The `extract_parts_of_speech` method parses the text in each element and extracts the words that are a given part of speech. For instance, to find all instances of adjectives:

```

parts_of_speech = graphlab.SFrame()
parts_of_speech['adjectives'] = graphlab.text_analytics.extract_parts_of_speech(sf['X1'],
parts_of_speech

```

```
Columns:
adjectives  dict

Rows: 10

Data:
+-----+
|      adjectives      |
+-----+
| {'ADJ': {'first': 1, 'nati... | 
| {'ADJ': {'first': 2, 'tech... | 
| {'ADJ': {'standard': 2, 'm... | 
| {'ADJ': {'standard': 8, 'p... | 
| {'ADJ': {'arsenopyrite': 2... | 
| {'ADJ': {'main': 6, 'roman... | 
| {'ADJ': {'third': 2, 'cruc... | 
| {'ADJ': {'main': 2, 'ethni... | 
| {'ADJ': {'first': 1, 'whic... | 
| {'ADJ': {'aseptic': 1, 'ri... | 
+-----+
[72269 rows x 1 columns]
```

Note that this API requires spaCy to be [installed](#).

Sentence Splitting

For an SArray of strings, where each row is assumed to be a natural English language document, the sentence splitter splits by sentence and outputs a list of sentences. This aids in analysis at the sentence level. For example, you may want a sentiment score for each sentence in a document. The following command accomplishes this for you:

```
sentences = graphlab.SFrame()
sentences['sent'] = graphlab.text_analytics.split_by_sentence(sf['X1'])
sentences
```

```
Columns:
  sent  list

Rows: 10

Data:
+-----+
|          sent          |
+-----+
| [alainconnes alain connes ... | 
| [americannationalstandards... | 
| [alberteinsteinstein near the b... | 
| [austriangerman as german ... | 
| [arsenic arsenic is a meta... | 
| [alps the alps alpen alpi ... | 
| [alexiscarrel born in sain... | 
| [adelaide adelaide is a co... | 
| [artist an artist is a per... | 
| [abdominalsurgery the thre... | 
+-----+
[72269 rows x 1 columns]
```

Note that this API requires spaCy to be [installed](#).

Topic Models

"Topic models" are a class of statistical models for text data. These models typically assume documents can be described by a small set of topics, and there is a probability of any word occurring for a given "topic".

For example, suppose we are given the documents shown below, where the first document begins with the text "The burrito was terrible. I..." and continues with a long description of the eater's woes. A topic model attempts to do two things:

1. Learn "topics": collections of words that co-occur in a meaningful way (as shown by the colored word clouds).
2. Learn how much each document pertains to each topic. This is represented by the colored circles, where larger circles indicate larger probabilities.



There are many variations of topic models that incorporate other sources of data, and there are a variety of algorithms for learning the parameters of the model from data. This section focuses on creating and using topic models with GraphLab Create.

Creating a model

The following example includes the SArray of documents that we created in previous sections: Each element represents a document in "bag of words" representation -- a dictionary with word keys and whose values are the number of times that word occurred in the document. Once in this form, it is straightforward to learn a topic model.

```
# Download data if you haven't already
import graphlab as gl
import os

if os.path.exists('wikipedia_w0'):
    docs = gl.SFrame('wikipedia_w0')
else:
    docs = gl.SFrame.read_csv('https://static.turi.com/datasets/wikipedia/raw/w0.csv', he
docs.save('wikipedia_w0')

# Remove stopwords and convert to bag of words
docs = gl.text_analytics.count_words(docs['X1'])
docs = docs.dict_trim_by_keys(gl.text_analytics.stopwords(), exclude=True)

# Learn topic model
model = gl.topic_model.create(docs)
```



There are a variety of additional arguments available which are covered in the [API Reference](#). The two most commonly used arguments are:

- `num_topics` : Changes the number of topics to learn.
- `num_iterations` : Changes how many iterations to perform.

The returned object is a `TopicModel` object, which exposes several useful methods. For example, `graphlab.topic_model.TopicModel.get_topics()` returns an SFrame containing the most probable words for each topic and a score related to how high that word ranks for that topic.

You may get details on a subset of topics by supplying a list of topic names or topic indices, as well as restrict the number of words returned per topic.

```
print model.get_topics()
```

```
+-----+-----+
| topic | word   |      score    |
+-----+-----+
| 0     | team   | 0.0116096428466 |
| 0     | state  | 0.00781031634036 |
| 0     | league | 0.00734010266384 |
| 0     | world  | 0.00707611127116 |
| 0     | party  | 0.00693101676527 |
| 1     | area   | 0.00939951828831 |
| 1     | film   | 0.00903846641308 |
| 1     | album  | 0.00810524480636 |
| 1     | states | 0.00592481420794 |
| 1     | part   | 0.00591876305919 |
+-----+-----+
[50 rows x 3 columns]
Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.
```

If we just want to see the top words per topic, this code snippet will rearrange the above SFrame to do that.

```
print model.get_topics(output_type='topic_words')
```

```
+-----+
|          words           |
+-----+
| [team, state, league, worl... |
| [area, film, album, states... |
| [city, family, war, built,... |
| [united, american, state, ... |
| [school, year, high, stati... |
| [season, music, church, ye... |
| [de, century, part, includ... |
| [university, 2006, line, s... |
| [age, population, people, ... |
| [time, series, work, life, ... |
+-----+
```

The model object keeps track of various useful statistics about how the model was trained and its current status.

```
model
```

```

Topic Model
Data:
    Vocabulary size:      632779
Settings:
    Number of topics:     10
    alpha:                5.0
    beta:                 0.1
    Iterations:           10
    Training time:        7.2635
    Verbose:               False
Accessible fields:
    m['topics']           An SFrame containing the topics.
    m['vocabulary']       An SArray containing the topics.
Useful methods:
    m.get_topics()         Get the most probable words per topic.
    m.predict(new_docs)    Make predictions for new documents.

```

To predict the topic of a given document, one can get an SArray of integers containing the most probable topic ids:

```
pred = model.predict(docs)
```

Combining the above method with standard SFrame capabilities, one can use predict to find documents related to a particular topic

```
docs_in_topic_0 = docs[model.predict(docs) == 0]
```

or join with other data in order to analyze an author's typical topics or how topics change over time. For example, if we had author and timestamp data, we could do the following:

```

>>> doc_data.column_names()
['timestamp', 'author', 'text']
>>> model = topic_model.create(doc_data['text'])
>>> doc_data['topic'] = m.predict(doc_data['text'])
>>> doc_data['author'][doc_data['topic'] == 1] # authors of docs in topic 1

```

Sometimes you want to know how certain the model's predictions are. One can optionally also get the probability of each topic for a set of documents. Each element of the returned SArray is a vector containing the probability of each document.

```

>>> pred = model.predict(docs, output_type='probability')
>>> pred
Out[13]: array('d', [0.049204587495375506, 0.14502404735479096, 0.028116907140214576, 0.1

```

Working with TopicModel objects

The value for each metadata field listed in `m.get_fields()` is accessible via `m[field]`.

A topic model will learn parameters for every word that is present in the corpus. The "vocabulary" stored by the model will return this list of words.

```
model['vocabulary']
```

```
dtype: str
Rows: 632779
['definitional', 'diversity', 'countereconomics', 'level', 'simultaneously', 'process', '
```

Similarly, we can obtain the current parameters for each word by requesting the "topics" stored by the model. Each element of the "topic_probabilities" column contains an array with length `num_topics` where element `k` is the probability of that word under topic `k`.

```
model['topics']
```

```
Columns:
topic_probabilities    array
vocabulary            str
```

```
Rows: 632779
```

```
Data:
```

topic_probabilities	vocabulary
[8.49255529868e-08, 8.6181...]	definitional
[8.49255529868e-08, 7.8347...]	diversity
[8.49255529868e-08, 7.8347...]	countereconomics
[8.49255529868e-08, 7.8347...]	level
[8.49255529868e-08, 7.8347...]	simultaneously
[8.49255529868e-08, 7.8347...]	process
[8.49255529868e-08, 7.8347...]	technology
[8.49255529868e-08, 2.4287...]	phenomena
[2.13163137997e-05, 0.0001...]	attitudes
[8.49255529868e-08, 2.8283...]	consumer

As with other models in GraphLab Create, it's also easy to save and load models.

```
model.save('my_model')
new_model = graphlab.load_model('my_model')
```

Importing from other formats

In some cases your data may be in a format that some refer to as "docword", where each row in the text file contains a document id, a word id, and the number of times that word occurs in that document. For this situation, check out the `parse_docword` utility:

```
docs = graphlab.text_analytics.parse_docword(doc_word_file, vocab_file)
```

Initializing from other models

It is also easy to create a new topic model from an old one – whether it was created using GraphLab Create or another package.

```
new_model = graphlab.topic_model.create(docs,
                                         num_topics=m['num_topics'],
                                         initial_topics=m['topics'])
```

Seeding the model with prior knowledge

To manually fix several words to always be assigned to a topic, use the `associations` argument. This can be useful for experimentation purposes:

For example, the following will ensure that "season" and "club" will have a high probability under topic 1 and "2008" and "2009" will have a high probability under topic 2:

```
associations = graphlab.SFrame({'word':['season', 'club', '2008', '2009'],
                                 'topic': [1, 1, 2, 2]})
```

If we fit a topic model using this option, we indeed find that "season" and "club" are present in topic 1, and we find other related words in the same topic.

```
m2 = graphlab.topic_model.create(docs,
                                  num_topics=20,
                                  num_iterations=50,
                                  associations=associations,
                                  verbose=False)
```

```
>>> m2.get_topics(num_words=10, output_type='topic_words')['words'][1]
['season',
 'club',
 'league',
 'played',
 'won',
 'football',
 'world',
 'cup',
 'year',
 'championship']
>>> m2.get_topics(num_words=10, output_type='topic_words')['words'][2]
['team',
 '2008',
 'game',
 '2009',
 '2007',
 '2010',
 'final',
 'player',
 '1',
 'players']
```

Evaluating topic models

A common quantitative way to evaluate topic models is to split each document into a training set and a test set, learn a topic model on the training portion of each document, and compute the probability of the heldout word counts under the model. A slight variation of this probability is called "perplexity". Lower values are better. Estimates of this quantity are provided during training. See [graphlab.text_analytics.util.random_split](#), [graphlab.text_analytics.util.perplexity](#), [TopicModel.evaluate](#) for helper functions to do this sort of evaluation on trained models.

A common way to qualitatively evaluate topic models is to examine the most probable words in each topic and count the number of words that do not fit with the rest. If there are topics with words that do not co-occur in your corpus, you may want to try:

- removing stop words and other words that are not interesting to your analysis
- changing the number of topics
- increasing the number of iterations

To learn more check out the [API Reference](#).

Evaluating models.

When evaluating models, choice of evaluation metrics is tied to the specific machine learning task. For example, if you built a classifier to detect spam emails vs. normal emails, then you should consider classification performance metrics, such as average accuracy, log-loss, and AUC. If you are trying to predict a score, such as Google's daily stock price, then you might want to consider regression metrics like the root mean-squared error (RMSE). If you are ranking items by relevance to a query, such as in a search engine, then you'll want to look into ranking losses such as precision-recall (also popular as a classification metric), or NDCG. These are all examples of task-specific performance metrics.

Regression Metrics

In a regression task, the model learns to predict numeric scores. An example is predicting the price of a stock on future days given past price history and other information about the company and the market. Another example is personalized recommendations, where the goal is to predict a user's rating for an item.

Here are a couple ways of measuring regression performance:

- [Root-Mean-Squared-Error](#)
- [Max-Error](#)

Classification Metrics

Classification is about predicting class labels given input data. In binary classification, there are two possible output classes. In multi-class classification, there are more than two possible classes. An example of binary classification is spam detection, where the input data could be the email text and metadata (sender, sending time), and the output label is either "spam" or "not spam." Sometimes, people use generic names for the two classes: "positive" and "negative," or "class 1" and "class 0."

There are many ways of measuring classification performance:

- [Accuracy](#)
- [Confusion matrix](#)
- [Log-loss](#)
- [Precision and Recall](#)
- [F-Scores](#)

- Receiver operating characteristic (ROC) curve
- Area under curve (AUC) ("curve" corresponds to the ROC curve)

Regression Metrics

In a regression task, the model learns to predict numeric scores. An example is predicting the price of a stock on future days given past price history and other information about the company and the market.

This section will deal with two ways of measuring regression performance:

- Root-Mean-Squared-Error
- Max-Error

Root-Mean-Squared-Error (RMSE)

The most commonly used metric for regression tasks is RMSE (Root Mean Square Error). This is defined as the square root of the average squared distance between the actual score and the predicted score:

$$\text{rmse} = \sqrt{\frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{n}}$$

Here, y_i denotes the true score for the i -th data point, and \hat{y}_i denotes the predicted value. One intuitive way to understand this formula is that it is the Euclidean distance between the vector of the true scores and the vector of the predicted scores, averaged by \sqrt{n} , where n is the number of data points.

```
import graphlab as gl

y      = gl.SArray([3.1, 2.4, 7.6, 1.9])
yhat = gl.SArray([4.1, 2.3, 7.4, 1.7])

print gl.evaluation.rmse(y, yhat)
```

```
0.522015325446
```

Max-Error

While RMSE is the most common metric, it can be hard to interpret. One alternative is to look at quantiles of the distribution of the absolute percentage errors. The Max-Error metric is the **worst case** error between the predicted value and the true value.

```
import graphlab as gl

y      = gl.SArray([3.1, 2.4, 7.6, 1.9])
yhat = gl.SArray([4.1, 2.3, 7.4, 1.7])

print gl.evaluation.max_error(y, yhat)
```

```
1.0
```

Classification Metrics

As mentioned previously, evaluation metrics are tied to the machine learning task. In this section, we will cover metrics for classification tasks. In binary classification, there are two possible output classes. In multi-class classification, there are more than two possible classes.

There are many ways of measuring classification performance:

- Accuracy
- Confusion matrix
- Log-loss
- Precision and Recall
- F-Scores
- Receiver operating characteristic (ROC) curve
- Area under curve (AUC) ("curve" corresponds to the ROC curve)

Accuracy

Accuracy simply measures how often the classifier makes the correct prediction. It's the ratio between the number of correct predictions and the total number of predictions (the number of test data points).

$$\text{accuracy} = \frac{\# \text{ correct}}{\# \text{ predictions}}$$

```
import graphlab as gl

y      = gl.SArray(["cat", "dog", "cat", "cat"])
yhat = gl.SArray(["cat", "dog", "cat", "dog"])

print gl.evaluation.accuracy(y, yhat)
```

```
0.75
```

Accuracy looks easy enough. However, it makes no distinction between classes; correct answers for each class are treated equally. Sometimes this is not enough. You might want to look at how many examples failed for each class. This would be the case if the cost of misclassification is different, or if you have a lot more test data of one class than the other.

For instance, making the call that a patient has cancer when he doesn't (known as a false positive) has very different consequences than making the call that a patient doesn't have cancer when he does (a false negative).

Multiclass Averaging

The multi-class setting is an extension of the binary setting. The accuracy metrics can be "averaged" across all the classes in many possible ways. Some of them are:

- **micro**: Calculate metrics globally by counting the total number of times each class was correctly predicted and incorrectly predicted.
- **macro**: Calculate metrics for each "class" independently, and find their unweighted mean. This does not take label imbalance into account.
- **None**: Return a metric corresponding to each class.

```
import graphlab as gl

y      = gl.SArray(["cat", "dog", "foosa", "cat"])
yhat = gl.SArray(["cat", "dog", "cat", "dog"])

print gl.evaluation.accuracy(y, yhat, average = "micro")
print gl.evaluation.accuracy(y, yhat, average = "macro")
print gl.evaluation.accuracy(y, yhat, average = None)
```

```
0.5
0.6666666666667
{'dog': 0.75, 'foosa': 0.75, 'cat': 0.5}
```

In general, when there are different numbers of examples per class, the average per-class accuracy will be different from the micro-average accuracy. When the classes are imbalanced, i.e., there are a lot more examples of one class than the other, then the accuracy will give a very distorted picture, because the class with more examples will dominate the statistic. In that case, you should look at the average per-class accuracy (average="micro"), as well as the individual per-class accuracy numbers (average=None). Per-class accuracy is not without its own caveats, however: for instance, if there are very few examples of one class, the test statistics for that class will be unreliable (i.e., they have large variance), so it's not statistically sound to average quantities with different degrees of variance.

Confusion matrix

A confusion matrix (or confusion table) shows a more detailed breakdown of correct and incorrect classifications for each class. Here is an example of how the confusion matrix can be computed.

The confusion table is an SFrame consisting of three columns:

- **target_label**: The label of the ground truth.
- **predicted_label**: The predicted label.
- **count**: The number of times the `target_label` was predicted as the `predicted_label`.

```
y      = gl.SArray(["cat", "dog", "foosa", "cat"])
yhat = gl.SArray(["cat", "dog", "cat", "dog"])

cf_matrix = gl.evaluation.confusion_matrix(y, yhat)
```

```
Columns:
  target_label    str
  predicted_label    str
  count        int

Rows: 4

Data:
+-----+-----+-----+
| target_label | predicted_label | count |
+-----+-----+-----+
|   foosa     |       cat      |   1   |
|   cat       |       dog      |   1   |
|   dog       |       dog      |   1   |
|   cat       |       cat      |   1   |
+-----+-----+-----+
[4 rows x 3 columns]
```

Looking at the matrix, one can clearly get a better picture of which class the model best identifies. This information is lost if one only looks at the overall accuracy.

Log-loss

Log-loss, or logarithmic loss, gets into the finer details of a classifier. In particular, if the raw output of the classifier is a numeric probability instead of a class label, then log-loss can be used. The probability essentially serves as a gauge of confidence. If the true label is "0" but the classifier thinks it belongs to class "1" with probability 0.51, then the classifier would be

making a mistake. But it's a near miss because the probability is very close to the decision boundary of 0.5. Log-loss is a "soft" measurement of accuracy that incorporates this idea of probabilistic confidence.

Mathematically, log-loss for a binary classifier looks like this:

$$\text{log-loss} = \sum_{i=1}^N y_i \log(p_i) + (1 - y_i) \log(1 - p_i)$$

Here, p_i is the probability that the i -th data point belongs to class "1", as judged by the classifier. y_i is the true label and is either "0" or "1". The beautiful thing about this definition is that it is intimately tied to information theory: Intuitively, log-loss measures the unpredictability of the "extra noise" that comes from using a predictor as opposed to the true labels. By minimizing the cross entropy, we maximize the accuracy of the classifier.

Here is an example of how this is computed:

```
import graphlab as gl

targets = gl.SArray([0, 1, 1, 0])
predictions = gl.SArray([0.1, 0.35, 0.7, 0.99])

log_loss = gl.evaluation.log_loss(targets, predictions)
```

1.5292569425208318

Logloss is undefined when a probability value $p_i = 0$, or $p_i = 1$. Hence, probabilities are clipped to $\max(\epsilon, \min(1 - \epsilon, p_i))$ where $\epsilon = 1.0 \times 10^{-15}$

Multi-class log-loss

In the multi-class setting, log-loss requires a vector of probabilities (that sum to 1) for each class label in the input dataset. In this example, there are three classes [0, 1, 2], and the vector of probabilities correspond to the probability of prediction for each of the three classes (while maintaining ordering).

```
targets      = gl.SArray([ 1, 0, 2, 1])
predictions = gl.SArray([[.1, .8, 0.1],
                       [.9, .1, 0.0],
                       [.8, .1, 0.1],
                       [.3, .6, 0.1]])

log_loss = gl.evaluation.log_loss(targets, predictions)
```

```
0.785478695933018
```

For multi-class classification, when the target label is of type **string**, then the probability vector is assumed to be a vector of probabilities of class as sorted alphanumerically. Hence, for the probability vector [0.1, 0.2, 0.7] for a dataset with classes ["cat", "dog", "rat"; the 0.1 refers to "cat", 0.2 refers to "dog", and 0.7 to "rat".

```
target      = gl.SArray([ "dog", "cat", "foosa", "dog"])
predictions = gl.SArray([[.1, .8, 0.1],
                        [.9, .1, 0.0],
                        [.8, .1, 0.1],
                        [.3, .6, 0.1]])
log_loss = gl.evaluation.log_loss(y, yhat)
```

```
1.5292569425208318
```

Precision & Recall

Precision and recall are actually two metrics. But they are often used together. Precision answers the question: *Out of the items that the classifier predicted to be true, how many are actually true?* Whereas, recall answers the question: *Out of all the items that are true, how many are found to be true by the classifier?*

The precision score quantifies the ability of a classifier to not label a **negative** example as **positive**. The precision score can be interpreted as the probability that a **positive** prediction made by the classifier is **positive**. The score is in the range [0,1] with 0 being the worst, and 1 being perfect.

The precision and recall scores can be defined as:

$$\text{precision} = \frac{\# \text{ true positive}}{\# \text{true positive} + \# \text{false positive}}$$

$$\text{recall} = \frac{\# \text{ true positive}}{\# \text{true positive} + \# \text{false negative}}$$

```
targets = graphlab.SArray([0, 1, 0, 0, 0, 1, 0, 0])
predictions = graphlab.SArray([1, 0, 0, 1, 0, 1, 0, 1])

pr_score = graphlab.evaluation.precision(targets, predictions)
rec_score = graphlab.evaluation.recall(targets, predictions)
print pr_score, rec_score
```

```
0.25, 0.5
```

Precision can also be defined then the target classes are of type **string**. For binary classification, when the target label is of type **string**, then the labels are sorted alphanumerically and the largest label is considered the "positive" label.

```
targets = graphlab.SArray(['cat', 'dog', 'cat', 'cat', 'cat', 'dog', 'cat', 'cat'])
predictions = graphlab.SArray(['dog', 'cat', 'cat', 'dog', 'cat', 'dog', 'cat', 'dog'])

pr_score = graphlab.evaluation.precision(targets, predictions)
rec_score = graphlab.evaluation.recall(targets, predictions)
print pr_score, rec_score
```

```
0.25, 0.5
```

Multi-class precision-recall

Precision and recall scores can also be defined in the multi-class setting. Here, the metrics can be "averaged" across all the classes in many possible ways. Some of them are:

- **micro**: Calculate metrics globally by counting the total number of times each class was correctly predicted and incorrectly predicted.
- **macro**: Calculate metrics for each "class" independently, and find their unweighted mean. This does not take label imbalance into account.
- **None**: Return the accuracy score for each class.corresponding to each class.

```
targets = graphlab.SArray(['cat', 'dog', 'cat', 'cat', 'cat', 'dog', 'cat', 'foosa'])
predictions = graphlab.SArray(['dog', 'cat', 'cat', 'dog', 'cat', 'dog', 'cat', 'foosa'])

macro_pr = graphlab.evaluation.precision(targets, predictions, average='macro')
micro_pr = graphlab.evaluation.precision(targets, predictions, average='micro')
per_class_pr = graphlab.evaluation.precision(targets, predictions, average=None)

print macro_pr, micro_pr
print per_class_pr
```

```
0.694444444444 0.625
{'foosa': 1.0, 'dog': 0.3333333333333333, 'cat': 0.75}
```

Note: The micro average precision, recall, and accuracy scores are mathematically equivalent.

Undefined Precision-Recall The precision (or recall) score is not defined when the number of true positives + false positives (true positives + false negatives) is zero. In other words, then the denominators of the respective equations are 0, the metrics are not defined. In those settings, we return a value of `None`. In the multi-class setting, the `None` is skipped during averaging.

F-scores (F1, F-beta)

The F1-score is a single metric that combines both precision and recall via their harmonic mean:

$$F_1 = 2 \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

The score lies in the range [0,1] with 1 being ideal and 0 being the worst. Unlike the arithmetic mean, the harmonic mean tends toward the smaller of the two elements. Hence the F1 score will be small if either precision or recall is small.

The F1-score (sometimes known as the balanced F-beta score), is a special case of a metric known as the F-Beta score, which *measures the effectiveness of retrieval with respect to a user who attaches β times as much importance to recall as to precision.*

$$F_\beta = (1 + \beta^2) \frac{\text{precision} * \text{recall}}{\beta^2 \text{precision} + \text{recall}}.$$

```
targets = graphlab.SArray([0, 1, 0, 0, 0, 1, 0, 0])
predictions = graphlab.SArray([1, 0, 0, 1, 0, 1, 0, 1])

f1      = graphlab.evaluation.f1_score(targets, predictions)
fbeta = graphlab.evaluation.fbeta_score(targets, predictions, beta = 2.0)
print f1, fbeta
```

```
0.333333333333 0.416666666667
```

Like the other metrics, the F1-score (or F-beta score) can also be defined when the target classes are of type **string**. For binary classification, when the target label is of type **string**, then the labels are sorted alphanumerically and the largest label is considered the "positive" label.

```

targets = graphlab.SArray(['cat', 'dog', 'cat', 'cat', 'cat', 'dog', 'cat', 'cat'])
predictions = graphlab.SArray(['dog', 'cat', 'cat', 'dog', 'cat', 'dog', 'cat', 'dog'])

f1    = graphlab.evaluation.f1_score(targets, predictions)
fbeta = graphlab.evaluation.fbeta_score(targets, predictions, beta = 2.0)
print f1, fbeta

```

0.333333333333 0.4166666666667

Multi-class F-scores

F-scores can also be defined in the multi-class setting. Here, the metrics can be "averaged" across all the classes in many possible ways. Some of them are:

- **micro**: Calculate metrics globally by counting the total number of times each class was correctly predicted and incorrectly predicted.
- **macro**: Calculate metrics for each "class" independently, and find their unweighted mean. This does not take label imbalance into account.
- **None**: Return the accuracy score for each class.corresponding to each class.

Note: The micro average precision, recall, and accuracy scores are mathematically equivalent.

Receiver Operating Characteristic (ROC Curve)

In statistics, a receiver operating characteristic (ROC), or ROC curve, is a graphical plot that illustrates the performance of a binary classifier system as its prediction **threshold** is varied. The ROC curve provides nuanced details about the behavior of the classifier. The curve is created by plotting the **true positive rate** (TPR) against the **false positive rate** (FPR) at various threshold settings.

This exotic sounding name originated in the 1950s from radio signal analysis, and was made popular by a 1978 paper by Charles Metz called "Basic Principles of ROC analysis." The ROC curve shows the sensitivity of the classifier by plotting the rate of true positives to the rate of false positives. In other words, it shows you how many correct positive classifications can be gained as you allow for more and more false positives. The perfect classifier that makes no mistakes would hit a true positive rate of 100% immediately, without incurring any false positives. This almost never happens in practice.

A good ROC curve has a lot of space under it (because the true positive rate shoots up to 100% very quickly). A bad ROC curve covers very little area.

```
targets = graphlab.SArray([0, 1, 1, 0])
predictions = graphlab.SArray([0.1, 0.35, 0.7, 0.99])

roc_curve = graphlab.evaluation.roc_curve(targets, predictions)
```

Data:

threshold	fpr	tpr	p	n
0.0	1.0	1.0	2	2
1e-05	1.0	1.0	2	2
2e-05	1.0	1.0	2	2
3e-05	1.0	1.0	2	2
4e-05	1.0	1.0	2	2
5e-05	1.0	1.0	2	2
6e-05	1.0	1.0	2	2
7e-05	1.0	1.0	2	2
8e-05	1.0	1.0	2	2
9e-05	1.0	1.0	2	2

[100001 rows x 5 columns]

The result of the **roc curve** is a multi-column **SFrame** with the following columns:

- **tpr**: True positive rate, the number of true positives divided by the number of positives.
- **fpr**: False positive rate, the number of false positives divided by the number of negatives.
- **p**: Total number of positive values.
- **n**: Total number of negative values.
- **class**: Reference class for this ROC curve (for multi-class classification).

Note: The ROC curve is computed using a binned histogram and hence always contains 100K rows. The binned histogram provides a curve that is accurate to the 5th decimal.

```
targets = graphlab.SArray(["cat", "dog", "cat", "dog"])
predictions = graphlab.SArray([0.1, 0.35, 0.7, 0.99])

roc_curve = graphlab.evaluation.roc_curve(targets, predictions)
```

```
Data:
+-----+-----+-----+-----+
| threshold | fpr | tpr | p | n |
+-----+-----+-----+-----+
| 0.0 | 1.0 | 1.0 | 2 | 2 |
| 1e-05 | 1.0 | 1.0 | 2 | 2 |
| 2e-05 | 1.0 | 1.0 | 2 | 2 |
| 3e-05 | 1.0 | 1.0 | 2 | 2 |
| 4e-05 | 1.0 | 1.0 | 2 | 2 |
| 5e-05 | 1.0 | 1.0 | 2 | 2 |
| 6e-05 | 1.0 | 1.0 | 2 | 2 |
| 7e-05 | 1.0 | 1.0 | 2 | 2 |
| 8e-05 | 1.0 | 1.0 | 2 | 2 |
| 9e-05 | 1.0 | 1.0 | 2 | 2 |
+-----+-----+-----+-----+
[100001 rows x 5 columns]
```

For binary classification, when the target label is of type **string**, then the labels are sorted alphanumerically and the largest label is chosen as the the **positive class**. The ROC curve can also be defined in the multi-class setting by returning a single curve for each class.

Area under the curve (AUC)

AUC stands for Area Under the Curve. Here, the curve is the ROC curve. As mentioned above, a good ROC curve has a lot of space under it (because the true positive rate shoots up to 100% very quickly). A bad ROC curve covers very little area.

```
targets = graphlab.SArray([0, 1, 1, 0])
predictions = graphlab.SArray([0.1, 0.35, 0.7, 0.99])

auc = graphlab.evaluation.auc(targets, predictions)
print auc
```

```
0.5
```

Note: The AUC score is computed using a binned histogram and hence always contains 100K rows. The binned histogram provides a curve that is accurate to the 5th decimal.

The AUC score can also be defined when the target classes are of type **string**. For binary classification, when the target label is of type **string**, then the labels are sorted alphanumerically and the largest label is considered the "positive" label.

```
targets = graphlab.SArray(["cat", "dog", "cat", "dog"])
predictions = graphlab.SArray([0.1, 0.35, 0.7, 0.99])

auc = graphlab.evaluation.auc(targets, predictions)
print auc
```

0.5

Multi-class area under curve

The AUC score can also be defined in the multi-class setting. Here, the metrics can be "averaged" across all the classes in many possible ways. Some of them are:

- **macro**: Calculate metrics for each "class" independently, and find their unweighted mean. This does not take label imbalance into account.
- **None**: Return a metric corresponding to each class.

Calibration curve

A calibration curve that compares the true probability of an event with its predicted probability. This chart shows if the trained model has well "calibrated" probability predictions. The closer the calibration curve is to the "ideal" curve, the more confident one can be in interpreting the probability predictions as a confidence score for predicting churn.

The graph is constructed by segmenting the probability space, from 0.0 to 1.0, into a fixed number of discrete steps. For instance, for a step size of 0.1, the probability space would be divided into 10 buckets: [0.0, 0.1), [0.1, 0.2) ... [0.9, 1.0]. Given a set of events for which a predicted probability has been computed, each predicted probability is rounded to the appropriate bucket. An aggregation is then performed to compute the total number of data points within each probability range along with fraction of those data points that were truly positive events. The calibration curve plots the fraction of events that were truly positive for each probability bucket. A perfectly calibrated model, the curve would be a line from (0.0, 0.0) to (1.0, 1.0).

Consider a simple example of a binary classification problem where a model was trained to predict a click or no-click event. If an event was “clicked”, and had a predicted probability of 0.345, using a binning of 0.1, the “number of events” for bin [0.3, 0.4) would be increased by one, and the “number of true” events would be increased by one. If an event with probability 0.231 was “not clicked”, the “number of events” for bin [0.2, 0.3) would be increased by one,

while the “number of true” events would not be increased. For each bin, the “number of true” over the “total number of events” becomes the observed probability (y-axis value). The average predicted probability for the bin becomes the x-axis value.

Precision-Recall Curve

The precision-recall curve plots the inherent trade-off between precision and recall. It is easier to understand and interpret this curve by understanding each of the components of this curve at different "thresholds".

Consider a simple example of a binary classification problem where a model was trained to predict a click or no-click event. If an event was “clicked”, Looking at the confusion matrix, the model predictions can be broken down into four categories:

- Events that were predicted to be "click" and were actually "click" events (True Positive, TP).
- Events that were predicted to be "not-click" but were not "click" events (False Positive, FP).
- Events that were predicted to be "not-click" and were actually "not-click" events (True Negative, TN).
- Events that were predicted to be "click" but were actually "not-click" events (False Negative, FN).

As defined earlier, precision is fraction of predicted click events that were actually click events while recall is the fraction of click events that were correctly predicted by the model. The definitions of precision and recall are not necessarily discrete events and can depend on the prediction probabilities. One can define a "threshold" of probability above which the model prediction can be considered a "click" prediction.

For example, a "threshold" of 0.5 implies that all probability predictions above 0.5 are considered "click" events and all predictions below are considered "not-click" events. By varying the threshold between 0.0 and 1.0, we can compute different precision and recall numbers. At a threshold of 0.0 all the model predictions are "click". At this point, the model exhibits perfect recall but has the worst possible precision. At a threshold of 1.0, the model predicts every event as "not-click"; this is perfect precision (the model was never wrong at predicting clicks!) but the worst possible recall.

This by plotting precision and recall at various thresholds, the precision-recall curve illustrates the trade-off of precision and recall for the model, at varying thresholds. Two models can be compared easily by plotting their precision-recall curve. A curve that is more to the top-right hand side of the plot represents a better model.

Model parameter search

Many machine learning models have hyper-parameters -- tuning knobs that can modify the behavior of the model. These are considered different than the rest of the model's parameters, as they may modify the nature of the model (e.g., the number of parameters to learn) or the training algorithm (e.g., the step size to use as training progresses). The settings of these parameters can be critical for optimizing the performance of a model for a specific source of data.

To perform model parameter search, you simply try out a variety of parameters and compare a model's performance. This chapter introduces a few key elements:

- **Models:** Which model are you trying to optimize? We discuss ways of optimizing GraphLab Create models, sklearn models, and your own custom functions.
- **Search space:** What is the set of parameters and range of each parameter that you want to consider? Below you'll hear about starting out a random search, gaining intuition with a grid search, and manually providing a list of parameters.
- **Evaluation:** How do you measure performance? We show examples of using validation sets and cross-validated metrics in order to measure the model's predictive ability given a set of parameters.
- **Execution:** Finally, many times this is an embarrassingly-parallel operation, and you can speed things up dramatically by distributing the work across many machines. We discuss how to do that below.

First we provide a quick example below, after which you can dig into each of the above categories in more detail.

Quick start

Suppose you have a simple dataset of (x,y) values:

```
import graphlab
from graphlab import model_parameter_search, SFrame
regression_data = SFrame({'x': range(100), 'y': [0,1] * (50)})
```

In the following example, we perform model parameter search over a linear regression model. We first split the data to create a training set that we will use for training the model, and a validation set to monitor the model's ability to generalize to unseen data.

```
training, validation = regression_data.random_split(0.8)
```

Next, we use the `model_parameter_search.create` method and pass in the data, a function that creates models, and a dictionary of parameter settings.

```
# Search over a grid of multiple hyper-parameters, with validation set
params = {'target': 'y'}
job = model_parameter_search.create((training, validation),
                                     graphlab.linear_regression.create,
                                     params)
```

You can query the returned object for results.

```
results = job.get_results()
results.column_names()
```

```
['model_id',
 'l1_penalty',
 'l2_penalty',
 'target',
 'training_rmse',
 'validation_rmse']
```

```
results[['l1_penalty', 'validation_rmse']]
```

l1_penalty	validation_rmse
10.0	0.499284926708
0.0001	0.496962825009
10.0	0.499284926708
1.0	0.497878172682
10.0	0.499284926708
10.0	0.499284293759
10.0	0.499284300085
0.1	0.497045347005
0.001	0.496981811586
0.0001	0.496962825009

In the above example, the model parameter search method does the following:

- Identifies the model (here linear regression) and retrieves a set of parameters to search.

For each parameter we provide an initial range of values to search.

- By default the method fits 10 models; this can be modified with the `max_models` argument.
- For each model, a random value of each parameter is chosen. This default search strategy can be called directly with `random_search.create`, and has been empirically shown to be quite effective [Bergstra, 2013].
- Each model is trained on the first element of the (training, validation) tuple. The model is then evaluated using the second element, representing `validation` data.
- The returned object can be queried asynchronously for the current status (`job.get_status()`) or the results (`job.get_results(wait=False)`).

To find out more details, continue to the next section about various supported models.

References

James Bergstra, Yoshua Bengio. [Random Search for Hyper-Parameter Optimization](#). JMLR 13 (2012).

Additional reading

Methodology for model parameter search is an active research area. Modern approaches often attempt to choose the next set of parameters based on the performance of previous parameter sets. This class of methods is often called Bayesian hyperparameter optimization. To learn more, the paper [Practical Bayesian Optimization of Machine Learning Algorithms](#) provides a great starting point.

Models

The `model_parameter_search.create` method has executed a search of parameters over a pre-defined space of possibilities. This can be helpful for newcomers who may not yet know the intricacies of each model and which parameters to consider in a first search.

The following GraphLab Create models have default search ranges provided:

- `kmeans.create`
- `logistic_classifier.create`
- `boosted_trees_classifier.create`
- `neuralnet_classifier.create`
- `svm_classifier.create`
- `linear_regression.create`
- `boosted_trees_regression.create`
- `ranking_factorization_recommender.create`
- `factorization_recommender.create`

If you are doing model parameter search for a scikit-learn model, we also have default search ranges for the following:

- `SVC`
- `LogisticRegression`
- `GradientBoostingClassifier`
- `GradientBoostingRegressor`
- `RandomForestClassifier`
- `RandomForestRegressor`
- `ElasticNet`
- `LinearRegression`

Suppose you want to specify your search space for a particular parameter. By specifying a set of values of `l2_penalty`, as we do below, the model search will only use `l2_penalty` values chosen from the provided list.

```
params = {'target': 'y', 'l2_penalty': [0.01, 0.05]}
job = model_parameter_search.create((training, validation),
                                    graphlab.linear_regression.create,
                                    params)
```

Tuning a GraphLab Create model

First of all, let's grab the Iris dataset, rename the final column to be 'target', and create a random train/test split.

```
import graphlab as gl
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data'
data = gl.SFrame.read_csv(url, header=False)
data.rename({'X5': 'target'})
(train, valid) = data.random_split(.8)
```

To do a parameter search with a `BoostedTreesClassifier` we simply specify the data, the model, and the value of the target parameter as a dictionary:

```
params = {'target': 'target'}
j = gl.model_parameter_search.create((train, valid),
                                      gl.boosted_trees_classifier.create,
                                      params)
```

This will use some sensible default parameter ranges, fitting 10 models. In the following results table, notice that we have trained models for several values of `column_subsample`, `max_depth`, etc.

```
j.get_results()
```

```
Columns:
model_id      int
column_subsample    float
max_depth      int
max_iterations  int
min_child_weight  int
min_loss_reduction  int
row_subsample   float
step_size       float
target         str
training_accuracy  float
validation_accuracy  float

Rows: 10
```

Data:

model_id	column_subsample	max_depth	max_iterations	min_child_weight
9	0.9	6	10	8
8	1.0	10	100	1
1	1.0	4	100	8
0	1.0	10	100	16
3	1.0	10	100	2

	2		0.8		8		100		2	
	5		0.8		8		10		2	
	4		0.8		10		100		4	
	7		0.8		8		50		16	
	6		0.8		6		50		4	
+	-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+									
+	-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+									
	min_loss_reduction		row_subsample		step_size		target		training_accuracy	
+	-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+									
	1		0.9		1e-05		target		0.962264150943	
	1		0.9		1e-05		target		0.980582524272	
	10		0.9		0.1		target		0.981132075472	
	10		0.9		0.1		target		0.950980392157	
	1		1.0		0.0		target		0.376146788991	
	0		0.9		0.5		target		1.0	
	1		0.9		0.001		target		0.981308411215	
	1		1.0		1e-05		target		0.963302752294	
	1		0.9		0.0		target		0.377358490566	
	10		1.0		0.001		target		0.980952380952	
+	-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+									
+	-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+									
	validation_accuracy									
+	-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+									
	0.921052631579									
	0.947368421053									
	0.947368421053									
	0.947368421053									
	0.236842105263									
	0.947368421053									
	0.947368421053									
	0.947368421053									
	0.236842105263									
	0.947368421053									
+	-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+									

Since this search involves a random combination of parameters, the results may vary each time you execute the function.

Tuning a sklearn model

You may also perform model parameter search on a sklearn model. Consider creating a train/test split of the iris dataset (as done [here](#)):

```

import numpy as np
from sklearn import cross_validation
from sklearn import datasets
from sklearn import svm

iris = datasets.load_iris()
iris.data.shape, iris.target.shape

X_train, X_test, y_train, y_test = cross_validation.train_test_split(
    iris.data, iris.target, test_size=0.4, random_state=0)

```

In this case, both the train and test datasets must be a tuple of numpy matrices (X, y) representing the feature matrix and the target vector, respectively. This time, we use `grid_search.create` to perform a grid search which fits a model for all possible combinations of parameters.

```

data = ((X_train, y_train), (X_test, y_test))
params = {'kernel': 'linear', 'C': [0.5, .75, 1.0]}
j = gl.grid_search.create(data, svm.SVC, params)

```

Running this job we get the following results table. By default, `model_parameter_search` methods use a sklearn model's `score` function to make predictions for the training and validation datasets. These values are presented in the `training_score` and `validation_score` columns.

```
j.get_results()
```

```

Columns:
  model_id      int
  C            float
  kernel       str
  training_score  float
  validation_score   float

Rows: 3

Data:
+-----+-----+-----+-----+
| model_id |  C  | kernel | training_score | validation_score |
+-----+-----+-----+-----+
|    1    | 0.75 | linear | 0.988888888889 |  0.966666666667 |
|    0    | 0.5  | linear | 0.988888888889 |      0.95      |
|    2    | 1.0  | linear | 0.988888888889 |  0.966666666667 |
+-----+-----+-----+-----+
[3 rows x 5 columns]

```

To get the fitted model with `c=.5` we can query for the first element from the response of

```
j.get_models() :
```

```
j.get_models()[0]
```

```
SVC(C=0.5, cache_size=200, class_weight=None, coef0=0.0, degree=3, gamma=0.0,
kernel='linear', max_iter=-1, probability=False, random_state=None,
shrinking=True, tol=0.001, verbose=False)
```

Tuning your own custom model

You may also want to tune a custom function that contains your own model, such as one or more GraphLab Create models with additional preprocessing or postprocessing logic. For example, suppose we want to train an ensemble of two models: a boosted trees classifier and a logistic regression classifier. We first create a function that takes in a dataset (along with some parameters) and returns a scoring function. The scoring function computes a weighted average of the predictions between the two models, where the weight is determined by the `proportion` argument.

```
def ensemble(train, target, proportion=.5):
    m1 = gl.boosted_trees_classifier.create(train, target=target)
    m2 = gl.logistic_classifier.create(train, target=target)
    def score(test):
        yhat1 = m1.predict(test)
        yhat2 = m2.predict(test)
        yhat = proportion * yhat1 + (1-proportion) * yhat2
        return yhat
    return score
```

Next, we need to define a function that can evaluate the returned scoring function with respect to the training and/or validation datasets. Here we use the scoring function to make predictions on each dataset, and we evaluate the accuracy with respect to the true target values.

```
def custom_evaluator(scorer, train, valid):
    yhat_train = scorer(train)
    yhat_valid = scorer(valid)
    return {'train_acc': gl.evaluation.accuracy(train['target'], yhat_train),
            'valid_acc': gl.evaluation.accuracy(valid['target'], yhat_valid)}
```

Finally, we can perform a model parameter search over a chosen set of proportions. We pass in our custom function, our parameters, and our custom evaluator. We can again use the Iris data, where this time we are classifying whether or not each instance has the label "Iris-setosa":

```
data = gl.SFrame.read_csv(url, header=False)
data.rename({'X5': 'target'})
data['target'] = data['target'] == 'Iris-setosa'
(train, test) = data.random_split(.3)

params = {'target': 'target', 'proportion': [.3, .5, .7]}
j = gl.grid_search.create((train, test),
                           ensemble,
                           params,
                           evaluator=custom_evaluator)
```

Debugging model search jobs

All model search jobs will attempt to fit a single model prior to scheduling the full search. We have found this can help speed up development by exposing simple errors faster. You may disable this trial run by setting `perform_trial_run=False`. For example, suppose we use the wrong name for the target column:

```
params = {'target': 'label', 'proportion': [.3, .5, .7]}
j = gl.grid_search.create((train, test),
                           ensemble,
                           params,
                           evaluator=custom_evaluator)
```

In this case, the first trial job fails and you see a message like:

```
No valid results have been created from this search.
[WARNING] Trial run failed prior to launching model parameter search. Please check for e
```

You may then retrieve the message of the thrown exception. In this case we used an incorrect value for the `target` parameter.

```
j.get_metrics()['exception_message']
Out[47]:
dtype: str
Rows: 2
['Runtime Exception. Column name label does not exist.', None]
```


Defining the search

Turi Distributed supports several specific methods for doing model parameter search.

Using dictionaries to specify parameters

The way you specify the set of parameters over which to search is through a dictionary. The dictionary keys are the names of the parameters and the values are the parameter values. Any values that are str, int, or floats are treated as a list containing a single value.

For example, specifying `{"target": "y"}` means that "y" will be the chosen target every time the model is fit. There are some list-typed arguments; in particular, `features` is a list of features to be used in the model. If you want to search over a list-typed argument, you must provide an iterable over valid argument values. For example, using `{"features": [[["col_a"]], ["col_a", "col_b"]]]}` would search over the two feature sets. If you just wanted to use the same set of features for each model, you would do `{"features": [[["col_a"]]]}`.

Specifying a grid search

Grid searches are especially useful when you have a relatively small set of parameters over which to search.

You may define a grid of parameters by specifying the possible values for each parameter. The method `grid_search.create` will then train a model for each unique combination.

The collection of all combinations of valid parameter values defines a grid of model parameters that will be considered. For example, providing the following `params` dictionary

```
params = {'target': 'label',
          'step_size': 0.3,
          'features': [[['a']], ['a', 'b']],
          'max_depth': [.1, .2]}
```

will create the following set of combinations:

```
[{'target': 'label', 'step_size': 0.3, 'features': ['a'], 'max_depth': .1},
 {'target': 'label', 'step_size': 0.3, 'features': ['a'], 'max_depth': .2},
 {'target': 'label', 'step_size': 0.3, 'features': ['a', 'b'], 'max_depth': .1},
 {'target': 'label', 'step_size': 0.3, 'features': ['a', 'b'], 'max_depth': .2}]
```

Using a random search space

You may not always know which areas of a search space are most promising. In such situations, it can be useful to pick parameter combinations from random distributions. The top-level method, `model_parameter_search`, currently chooses `random_search.create` by default.

For example, for a real-valued parameter such as `step_size`, you could might want to draw values from an [exponential distribution](#). In the following example, each parameter combination will contain

- a `target` value of 'Y'
- a `max_depth` value of either 5 or 7 (chosen randomly)
- a `step_size` value drawn randomly from an exponential distribution with mean of 0.1

```
import scipy.stats
url = 'https://static.turi.com/datasets/xgboost/mushroom.csv'
data = gl.SFrame.read_csv(url)
data['label'] = (data['label'] == 'p')

train, valid = data.random_split(.8)
params = {'target': 'label',
          'max_depth': [5, 7],
          'step_size': scipy.stats.distributions.expon(.1)}
job = gl.random_search.create((train, valid),
                             gl.boosted_trees_regression.create,
                             params)
job.get_results()
```

```

Columns:
  model_id      int
  max_depth     int
  step_size     float
  target    str
  training_rmse float
  validation_rmse float

Rows: 8

Data:
+-----+-----+-----+-----+-----+
| model_id | max_depth | step_size | target | training_rmse |
+-----+-----+-----+-----+-----+
|    9    |      7    | 0.742280945789 | label | 0.000562821322042 |
|    8    |      5    | 0.37544111673 | label | 0.00963600115039 |
|    1    |      5    | 0.138909527035 | label | 0.11368970605 |
|    0    |      7    | 0.977843893103 | label | 0.000269710408328 |
|    3    |      7    | 0.32559648473 | label | 0.0110626696535 |
|    2    |      5    | 0.330703633987 | label | 0.0137912720349 |
|    5    |      7    | 0.408652318249 | label | 0.00367912426229 |
|    4    |      7    | 0.295146249231 | label | 0.0162840474088 |
+-----+-----+-----+-----+-----+
+-----+
| validation_rmse |
+-----+
| 0.000790839939725 |
| 0.0123972020261 |
| 0.114722098681 |
| 0.000369491390958 |
| 0.0120762185507 |
| 0.0169411827805 |
| 0.00439583387505 |
| 0.0171414864358 |
+-----+

```

Manually specifying parameters

If you want full control over your parameter search, then you can use the `manual_search.create` function. All you need to do is to pass in a list of parameter dictionaries; a model will be fit for each parameter set.

```

factory = gl.boosted_trees_classifier.create
params = [{'target': 'label', 'max_depth': 3},
          {'target': 'label', 'max_depth': 6}]
job = gl.manual_search.create((train, valid),
                             factory, params)

```


Cross Validation

Data is the first argument for all of the model parameter search functions. This argument allows for several different input types to allow you to better evaluate model performance on a given set of parameters.

You can provide a train/test pair: by default, each model will be trained on the first element and evaluated on both elements.

```
url = 'https://static.turi.com/datasets/xgboost/mushroom.csv'  
data = gl.SFrame.read_csv(url)  
(train, valid) = data.random_split(.7)  
gl.model_parameter_search.create((train, valid), my_model, my_params)
```

You can provide a list of train/test pairs. The results for each model will be averaged across the folds.

```
folds = [(train0, valid0), (train1, valid1)]  
gl.model_parameter_search.create(folds, my_model, my_params)
```

We also provide a convenience object [KFold](#) for performing model search using K folds.

```
folds = gl.cross_validation.KFold(sf, 5)  
job = gl.random_search.create(folds,  
                             my_model,  
                             my_params)
```

In this case, the returned [KFold](#) object splits the data lazily to minimize communication costs.

Cross validation for a single parameter set

We also provide a convenience function for evaluating model performance via cross validation for a given set of parameters.

```
url = 'https://static.turi.com/datasets/xgboost/mushroom.csv'
data = gl.SFrame.read_csv(url)
data['label'] = (data['label'] == 'p')
folds = gl.cross_validation.KFold(data, 5)
params = {'target': 'label', 'max_depth': 5}
job = gl.cross_validation.cross_val_score(folds,
                                         gl.boosted_trees_classifier.create,
                                         params)
print job.get_results()
```

This is analogous to sklearn's [cross_val_score](#).

Additional reading

To learn more about the benefits of k-fold cross-validation, check out Chapter 5.1 of [Introduction to Statistical Learning](#).

Distributing model parameter search

For all model parameter search methods and `cross_val_score`, you have the choice of running the jobs locally or remotely.

Locally

By default, jobs are scheduled to run locally in an asynchronous fashion. This is called a LocalAsync environment.

Remotely

You may also run jobs on an EC2 cluster or a Hadoop cluster. This is especially useful when you want to perform a larger scale parameter search.

To launch a job on an EC2 cluster, you first create an EC2 environment and pass it into the `environment` argument:

```
ec2config = graphlab.deploy.Ec2Config()
ec2 = graphlab.deploy.ec2_cluster.create(name='mps',
                                         s3_path='s3://bucket/path',
                                         ec2_config=ec2config,
                                         num_hosts=4)

j = graphlab.model_parameter_search.create((train, valid),
                                           my_model, my_params,
                                           environment=ec2)
```

For launching jobs on a Hadoop cluster, you instead create a Hadoop environment and pass this object into the `environment` argument:

```
hd = gl.deploy.hadoop_cluster.create(name='hadoop-cluster',
                                      turi_dist_path=<path to installation>

j = graphlab.model_parameter_search.create((train, valid),
                                           my_model, my_params,
                                           environment=hd)
```

For more details on creating EC2- and Hadoop-based environments, checkout the [API docs](#) or the [Deployment](#) chapter of the userguide.

When getting started, it is useful to keep `perform_trial_run=True` to make sure you are creating your models properly.

Applications

In this section, we demonstrate how you can use the Turi Machine Learning Platform to tackle these common scenarios:

- [Recommender systems](#)
- [Data matching](#)
- [Lead scoring](#)
- [Churn prediction](#)
- [Frequent Pattern Mining](#)
- [Sentiment analysis](#)
- [Anomaly detection](#)

Recommender systems

A recommender system allows you to provide personalized recommendations to users. With this toolkit, you can train a model based on past interaction data and use that model to make recommendations.

Note: Follow the steps in the [sample-movie-recommender](#) GitHub repository to get the code and data for this example.

Input data

Creating a recommender model typically requires a data set to use for training the model, with columns that contain the user IDs, the item IDs, and (optionally) the ratings.

```
>>> actions = gl.SFrame.read_csv('../dataset/ml-20m/ratings.csv')
+-----+-----+-----+
| userId | movieId | rating | timestamp |
+-----+-----+-----+
| 1      | 2       | 3.5   | 1112486027 |
| 1      | 29      | 3.5   | 1112484676 |
| 1      | 32      | 3.5   | 1112484819 |
| 1      | 47      | 3.5   | 1112484727 |
| 1      | 50      | 3.5   | 1112484580 |
| 1      | 112     | 3.5   | 1094785740 |
| 1      | 151     | 4.0   | 1094785734 |
| 1      | 223     | 4.0   | 1112485573 |
| 1      | 253     | 4.0   | 1112484940 |
| 1      | 260     | 4.0   | 1112484826 |
+-----+-----+-----+
```

For information on how to load data into an SFrame from other sources, see the chapter on [Loading and Saving SFrames](#).

You may have additional data about users or items. For example we might have a dataset of movie metadata.

```
>>> items = gl.SFrame.read_csv('./dataset/ml-20m/movies.csv')
+-----+-----+-----+
| movieId | title | genres | year |
+-----+-----+-----+
| 1 | Toy Story | [Adventure, Anim... | 1995 |
| 2 | Jumanji | [Adventure, Chil... | 1995 |
| 3 | Grumpier Old Men | [Comedy, Romance] | 1995 |
| 4 | Waiting to Exhale | [Comedy, Drama, ... | 1995 |
| 5 | Father of the Br... | [Comedy] | 1995 |
+-----+-----+-----+
```

If you have data like this associated with each item, you can build a model from just this data using the item content recommender. In this case, providing the user and item interaction data at training time is optional.

Building a model

There are a variety of machine learning techniques that can be used to build a recommender model. GraphLab Create provides a method `graphlab.recommender.create` that will automatically choose an appropriate model for your data set.

First we create a random split of the data to produce a validation set that can be used to evaluate the model.

```
training_data, validation_data = gl.recommender.util.random_split_by_user(actions, 'userId')
model = gl.recommender.create(training_data, 'userId', 'movieId')
```

Now that you have a model, you can make recommendations

```
# You can now make recommendations for all the users you've just trained on
results = model.recommend()
```

Learn more

The following sections provide more information about the recommender model:

- [Using trained models](#)
 - making recommendations
 - finding similar items and users
 - evaluating the model
 - interactive visualizations
 - saving models

- and more
- [Choosing a model](#)
 - data you might encounter (implicit or explicit)
 - types of models worth considering (item-based similarity, factorization-based models, and so on).
- [API Docs](#)

Using trained models

All recommender objects in the `graphlab.recommender` module expose a common set of methods, such as `recommend` and `evaluate`.

In this section we will cover

- [Making recommendations](#)
- [Finding similar items](#)
- [Visualizations](#)
- [Saving and loading](#)

Making recommendations

The trained model can now make recommendations of new items for users. To do so, call `model.recommend()` with an SArray of user ids. If `users` is set to None, then `model.recommend()` will make recommendations for all the users seen during training, automatically excluding the items that are observed for each user. In other words, if `data` contains a row "Alice, The Great Gatsby", then `model.recommend()` will not recommend "The Great Gatsby" for user "Alice". It will return at most `k` new items for each user, sorted by their rank. It will return fewer than `k` items if there are not enough items that the user has not already rated or seen.

The `score` column of the output contains the *unnormalized* prediction scores for each user-item pair. The semantic meanings of these scores may differ between models. For the linear regression model, for instance, a higher average score for a user means that the model thinks that this user is generally more enthusiastic than others.

Finding Similar Items

Many of the above models make recommendations based on some notion of similarity between a pair of items. Querying for similar items can help you understand the model's behavior on your data.

We have made this process very easy with the `get_similar_items` function:

```
similar_items = model.get_similar_items(my_list_of_items, k=20)
```

The above will return an SFrame containing the 20 nearest items for every item in `my_list_of_items`. The definition of "nearest" depends on the type of similarity used by the model. For instance, "jaccard" similarity measures the two item's overlapping users. The

'score' column contains a similarity score ranging between 0 and 1, where larger values indicate increasing similarity. The mathematical formula used for each type of similarity can be found in the API documentation for [ItemSimilarityRecommender](#).

For a factorization-based model, the similarity used for is the Euclidean distance between the items' two factors, which can be obtained using `m['coefficients']`.

Saving and loading models

The model can be saved for later use, either on the local machine or in an AWS S3 bucket. The saved model sits in its own directory, and can be loaded back in later to make more predictions.

```
# Save the model for later use
model.save("my_model")
```

Like other models in GraphLab Create, we can load the model back:

```
model = gl.load_model("my_model")
```

Visualizations

Once a model has been trained, you can easily visualize the model. There are three built-in visualizations to help explore, explain, and evaluate the model:

- **Explore:** This view helps explore the predictions (and associated recommendations) made by the model.
- **Evaluate:** This view helps quickly provide all the information needed to determine if the model compares well with a simple popularity-based baseline.
- **Overview:** This view combines the explore and evaluate views into a tabbed view.

For example, to create an the "Overview" tab, you launch the interactive visualization using the following code:

```
# Get the metadata ready
data_dir = './dataset/ml-20m/'
urls = gl.SFrame.read_csv(path.join(data_dir, 'movie_urls.csv'))
items = items.join(urls, on='movieId')
users = gl.SFrame.read_csv(path.join(data_dir, 'user_names.csv'))

# Interactively evaluate and explore recommendations
view = model.views.overview(validation_set=validation_data,
                             user_data=users,
                             user_name_column='name',
                             item_data=items,
                             item_name_column='title',
                             item_url_column='url')

view.show()
```

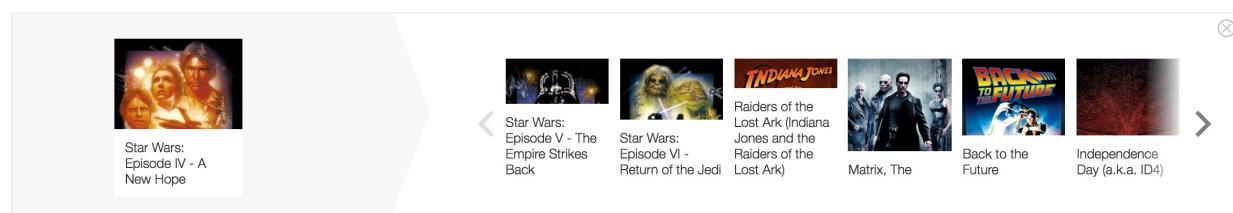
Explore

Enter a User or Item

|

Get Recommendations

Enter an item to get similar items, or enter a user to get personalized recommended items.



The explore tab provides an interactive model visualization that you can use on the recommender model you just built. In this screen you can explore the model's behavior and get explanations of the individual recommendations.

- You can search for a given item, and understand what recommendations would be made for a user who had only interacted with that item.
- For each shown item, you can understand what data associated with that particular recommendation
- You can also get recommendations for that item. Similarly you can get recommendations for a given user.

By understanding the qualitative aspects of the model you can build confidence in your model by

- monitoring how your model behaves in particular situations, detecting problems with your model prior to going to production,
- exposing ways you can improve your model in the future.

Choosing a Model

In this section, we give some intuition for which modeling choices you may make depending on your data and your task. Each recommender model in GraphLab Create has certain strengths that fit well with certain types of data and different objectives.

The easiest way to choose a model is to let GraphLab Create choose your model for you. This is done by simply using the default `recommender.create` function, which chooses the model based on the data provided to it. As an example, the following code creates a basic item similarity model and then generates recommendations for each user in the dataset:

```
m = graphlab.recommender.create(data, user_id='user', item_id='movie')
recs = m.recommend()
```

Using the default `create` method provides an excellent way to quickly get a recommender model up and running, but in many cases it's desirable to have more control over the process.

Effectively choosing and tuning a recommender model is best done in two stages. The first stage is to match the type of data up with the correct model or models, and the second stage is to correctly evaluate and tune the model(s) and assess their accuracy. Sometimes one model works better and sometimes another, depending on the data set. In a later section, we'll look at evaluating a model so you can be confident you chose the best one.

Data Type: Explicit, Implicit, or Item Content Data?

With *Explicit data*, there is an associated target column that gives a score for each interaction between a users and an items. An example of this type would be a data set of user's ratings for movies or books. With this type of data, the objective is typically to either predict which new items a user would rate highly or to predict a user's rating on a given item.

Implicit data does not include any rating information. In this case, a dataset may have just two columns -- user ID and item ID. For this type of data, the recommendations are based on which items are similar to the items a user has interacted with.

The third type of data that GraphLab Create can use to build a recommender system is *item content data*. In this case, information associated with each individual item, instead of the user interaction patterns, is used to recommend items similar to a collection of items in a query set. For example, item content could be a text description of an item, a set of key words, an address, categories, or even a list of similar items taken from another model.

Working with Explicit Data

If your data is *explicit*, i.e., the observations include an actual rating given by the user, then the model you wish to use depends on whether you want to predict the rating a user would give a particular item, or if you want the model to recommend items that it believes the user would rate highly.

If you have ratings data and care about accurately predicting the rating a user would give a specific item, then we typically recommend you use the `factorization_recommender`. In this model the observed ratings are modeled as a weighted combination of terms, where the weights (along with some of the terms, also known as factors) are learned from data. All of these models can easily incorporate user or item side features.

A linear model assumes that the rating is a linear combination of user features, item features, user bias, and item popularity bias. The `factorization_recommender` goes one step further and allows each rating to also depend on a term representing the inner product of two vectors, one representing the user's affinity to a set of latent preference modes, and one representing the item's affinity to these modes. These are commonly called latent factors and are automatically learned from observation data. When side data is available, the model allows for interaction terms between these learned latent factors and all the side features. As a rule of thumb, the presence of side data can make the model more finicky to learn (due to its power and flexibility).

If you care about *ranking performance*, instead of simply predicting the rating accurately, then choose [ItemSimilarityRecommender](#) or [RankingFactorizationRecommender](#). With rating data, the `item_similarity_recommender` model scores items based on how likely they predict the user will rate them highly, but the absolute values of the predicted scores may not match up with the actual ratings a user would give the item.

The [RankingFactorizationRecommender](#) tries to recommend items that are both similar to the items in a user's dataset and, if rating information is provided, those that would be rated highly by the user. It tends to predict ratings with less accuracy than the non-ranking `factorization_recommender`, but it tends to do much better at choosing items that a user would rate highly. This is because it also penalizes the predicted rating of items that are significantly different from the items a user has interacted with. In other words, it only predicts a high rating for user-item pairs in which it predicts a high rating and is confident in that prediction.

Furthermore, this model works particularly well when the target ratings are binary, i.e., if they come from thumbs up/thumbs down flags. In this case, use the input parameter `binary_targets = True`.

When a `target` column is provided, the model returned by the default `recommender.create` function is a matrix factorization model. The matrix factorization model can also be called directly with `ranking_factorization_recommender.create`. When using the model-specific `create` function, other arguments can be provided to better tune the model, such as `num_factors` or `regularization`. See the documentation on [RankingFactorizationRecommender](#) for more information.

```
m = graphlab.ranking_factorization_recommender.create(data,
                                                       user_id='user',
                                                       item_id='movie',
                                                       target='rating')
```

Working with Implicit Data

The goal a recommender system built with implicit data is to recommend items that are similar to those similar to the collection of items a user has interacted with. "Similar" in this case is determined by other user interactions -- if most users with similar behavior to a given user also interacted with a item the given user had not, that item would likely in the given user's recommendations.

In this case, the default `recommender.create` function in the example code above code returns an [ItemSimilarityRecommender](#) which computes the similarity between each pair of items and recommends items to each user that are closest to items they have already used or liked:

```
m = graphlab.item_similarity_recommender.create(data,
                                                 user_id='user',
                                                 item_id='movie')
```

The `ranking_factorization_recommender` is also great for implicit data, and can be called the same way:

```
m = graphlab.ranking_factorization_recommender.create(data,
                                                       user_id='user',
                                                       item_id='movie')
```

With implicit data, the ranking factorization model has two solvers, one which uses a randomized sgd-based method to tune the results, and the other which uses an implicit form of alternating least squares (iALS). The default sgd-based method samples unobserved items along with the observed ones, then treats them as negative examples. This is the default solver. Implicit ALS is a version of the popular Alternating Least Squares (ALS) algorithm that attempts to find factors that distinguish between the given user-item pairs and

all other negative examples. This algorithm can be faster than the sgd method, particularly if there are many items, but it does not currently support side features. This solver can be activated by passing in `solver = "ials"` to `ranking_factorization_recommender.create`. On some datasets, one of these solvers can yield better precision-recall scores than the `item_similarity_recommender`.

Item Content Data

The [ItemContentRecommender](#) builds a model similar to the item similarity model, but uses similarities between item content to actually build the model. In this model, the similarity score between two items is calculated by first computing the similarity between the item data for each column, then taking a weighted average of the per-column similarities to get the final similarity. The recommendations are generated according to the average similarity of a candidate item to all the items in a user's set of rated items. This model can be created without observation data about user-item interactions, in which case such information must be passed in at recommend time in order to make recommendations.

Note that in most situations, the similarity patterns of items can be inferred effectively from patterns in the user interaction data, and the `factorization_recommender` and `item_similarity_recommender` do this effectively. However, leveraging information about item content can be very useful, particularly when the user-item interaction data is sparse or not known until recommend time.

Side information for users, items, and observations

In many cases, additional information about the users or items can improve the quality of the recommendations. For example, including information about the genre and year of a movie can be useful information in recommending movies. We call this type of information user side data or item side data depending on whether it goes with the user or the item.

Including side data is easy with the `user_data` or `item_data` parameters to the `recommender.create()` function. These arguments are SFrames and must have a user or item column that corresponds to the `user_id` and `item_id` columns in the observation data. Internally, the data is joined to the particular user or item when training the model, the data is saved with the model and also used to make recommendations.

In particular, the [FactorizationRecommender](#) and the [RankingFactorizationRecommender](#) both incorporate the side data into the prediction through additional interaction terms between the user, the item, and the side feature. For the actual formula, see the API docs for the [FactorizationRecommender](#). Both of these models also allow you to obtain the parameters that have been learned for each of the side features via the `m['coefficients']` argument.

Side data may also be provided for each observation. For example, it might be useful to have recommendations change based on the time at which the query is being made. To do so, you could create a model using an SFrame that contains a time column, in addition to a user and item column. For example, a "time" column could include a string indicating the hour; this will be treated as a categorical variable and the model will learn a latent factor for each unique hour.

```
# sf has columns: user_id, item_id, time
m = gl.ranking_factorization_recommender.create(sf)
```

In order to include this information when requesting observations, you may include the desired additional data as columns in an SFrame for the `users` argument to `m.recommend()`. In our example above, when querying for recommendations, you would include the time that you want to use for each set of recommendations.

```
users_query = gl.SFrame({'user_id': [1, 2, 3], 'time': ['10pm', '10pm', '11pm']})
m.recommend(users=user_query)
```

In this case, recommendations for user 1 and 2 would use the parameters learned from observations that occurred at 10pm, whereas the recommendations for user 3 would incorporate parameters corresponding to 11pm. For more details, check out [recommend](#) in the API docs.

You may check the number of columns used as side information by querying

```
m['observation_column_names'] , m['user_side_data_column_names'] , and
m['item_side_data_column_names'] . By printing the model, you can also see this information.
In the following model, we had four columns in the observation data (two of which were
user_id and item_id ) and four columns in the SFrame passed to item_side_data (one of
which was item_id ):
```

```
Class : RankingFactorizationRecommender

Schema
-----
User ID : user_id
Item ID : item_id
Target : None
Additional observation features : 2
Number of user side features : 0
Number of item side features : 3
```

If new side data exists when recommendations are desired, this can be passed in via the `new_observation_data`, `new_user_data`, and `new_item_data` arguments. Any data provided there will take precedence over the user and item side data stored with the model.

Not all of the models make use of side data: the `popularity_recommender` and `item_similarity_recommender` create methods currently do not use it.

Suggested pre-processing techniques

Lastly, here are a couple of common data issues that can affect the performance of a recommender. First, if the observation data is very sparse, i.e., contains only one or two observations for a large number of users, then none of the models will perform much better than the simple baselines available via the `popularity_recommender`. In this case, it might help to prune out the rare users and rare items and try again. Also, re-examine the data collection and data cleaning process to see if mistakes were made. Try to get more observation data per user and per item, if you can.

Another issue often occurs when usage data is treated as ratings. Unlike explicit ratings that lie on a nice linear interval, say 0 to 5, usage data can be badly skewed. For instance, in the Million Song dataset, one user played a song more than 16,000 times. All the models would have a difficult time fitting to such a badly skewed target. The fix is to bucketize the usage data. For instance, any play count greater than 50 can be mapped to the maximum rating of 5. You can also clip the play counts to be binary, e.g., any number greater than 2 is mapped to 1, otherwise it's 0.

For more on this check out our blog post, [Choosing a Recommender Model](#).

Evaluating Model Performance

When trying out different recommender models, it's critical to have a principled way of evaluating their performance. The standard approach to this is to split the observation data into two parts, a training set and a test set. The model is trained on the training set, and then evaluated on the test set -- evaluating your model on the same dataset that it was trained on gives a very bad idea of how well it will perform in practice. Once the model type and associated parameters are chosen, the model can be trained on the full dataset.

With recommender systems, we can evaluate models using two different metrics, RMSE and precision-recall. RMSE measures how well the model predicts the score of the user, while precision-recall measures how well the `recommend()` function recommends items that the user also chooses. For example, the best RMSE value is when the model exactly predicts the value of all the ratings in the test set. Similarly, the best precision-recall value is when

the user has 5 items in the test set and recommend() recommends exactly those 5 items. While both can be important depending on the type of data and desired task, precision-recall is often more useful in evaluating how well a recommender system will perform in practice.

The GraphLab Create recommender toolkit includes a function, `gl.recommender.random_split_by_user`, to easily generate training and test sets from observation data. Unlike `gl.SFrame.random_split`, it only puts data for a subset of the users into the test set. This is typically sufficient for evaluating recommender systems.

`gl.recommender.random_split_by_user` generates a test set by first choosing a subset of the users at random, then choosing a random subset of that user's items. By default, it chooses 1000 users and, for each of these users, 20% of their items on average. Note that not all users may be represented by the test set, as some users may not have any of their items randomly selected for the test set.

Once training and test set are generated, the `gl.recommender.util.compare_models` function allows easy evaluation of several models using either RMSE or precision-recall. These models may be the same models with different parameters or completely different types of model.

The GraphLab Create recommender toolkit provides several ways of working with rating data while ensuring good precision-recall. To accurately evaluate the precision-recall of a model trained on explicit rating data, it's important to only include highly rated items in your test set as these are the items a user would likely choose. Creating such a test set can be done with a handful of SFrame operations and `gl.recommender.random_split_by_user` :

```
high_rated_data = data[data["rating"] >= 4]
low_rated_data = data[data["rating"] < 4]
train_data_1, test_data = gl.recommender.random_split_by_user(
    high_rated_data, user_id='user', item_id='movie')
train_data = train_data_1.append(low_rated_data)
```

Other examples of comparing models can be found in the API documentation for `gl.recommender.util.compare_models`.

Data Matching

Data matching is the identification and aggregation of data records that correspond to the same real-world entity. Often data matching problems arise when aggregating datasets from different sources, but the field of data matching encompasses several different tasks that have quite different data contexts. The GraphLab Create data matching toolkit provides four tools to help you quickly accomplish the most common data matching tasks.

Record linker is the most straightforward data matching task: linking structured query records to a fixed reference set, also in tabular form.

Deduplication also works with structured datasets, but differs from record linking in that there is no fixed reference dataset. Instead, all records from the input datasets are matched to each other, with duplicate records given the same entity label (akin to a cluster label). Deduplication examples include combining records about customers who sign up for a service multiple times, or aggregating location information about businesses obtained from multiple listing services.

Autotagging involves matching documents to a fixed set of tags, listed and described in a tabular dataset. Examples of this include finding product names in unstructured customer reviews or blog posts, and matching unstructured merchant product offers to a product catalog.

Similarity search takes high level data objects like images, documents, or even combinations of the two, and finds similar items in a reference set of items. Typically, implementing a system to accomplish this task requires substantial domain knowledge to convert the raw data objects into numeric vectors, followed by a nearest neighbors search.

Record linker

The record linker tool matches structured query records to a fixed set of reference records with the same schema. A common example of this is matching address records from a query dataset to a relatively error-free reference address database.

To illustrate this example with the record linker tool, we use synthetic address data generated by and packaged with the [FEBRL](#) program, another data matching tool. For the sake of illustration suppose the set called "refs" is a clean set of reference addresses (say, from a government collected and curated database), while the SFrame called "query" contains new data with many errors.

```
import os
import graphlab as gl

col_types = {'street_number': str, 'postcode': str}

if os.path.exists('febrl_F_org_5000.csv'):
    refs = gl.SFrame.read_csv('febrl_F_org_5000.csv',
                              column_type_hints=col_types)
else:
    url = 'https://static.turi.com/datasets/febrl_synthetic/febrl_F_org_5000.csv'
    refs = gl.SFrame.read_csv(url, column_type_hints=col_types)
    refs.save('febrl_F_org_5000.csv')

if os.path.exists('febrl_F_dup_5000.csv'):
    query = gl.SFrame.read_csv('febrl_F_dup_5000.csv',
                               column_type_hints=col_types)
else:
    url = 'https://static.turi.com/datasets/febrl_synthetic/febrl_F_dup_5000.csv'
    query = gl.SFrame.read_csv(url, column_type_hints=col_types)
    query.save('febrl_F_dup_5000.csv')
```

For simplicity, we'll discard several of the features right off the bat. The remaining features have the same schema in both the `refs` and `query` SFrames. Note the large number of errors present even in the first few rows of the query dataset.

```
address_features = ['street_number', 'address_1', 'suburb', 'state', 'postcode']
refs = refs[address_features]
query = query[address_features]

refs.head(5)
```

```
+-----+-----+-----+-----+
| street_number | address_1 | suburb | state | postcode |
+-----+-----+-----+-----+
|      95      | galway place | st marys |       | 2681   |
|      12      | burnie street | wycheproof | nsw  | 2234   |
|      16      | macrobertson street | branxton | qld  | 3073   |
|     170      | bonrook street | brighton east | nsw  | 3087   |
|      32      | proserpine court | helensvale | qld  | 2067   |
+-----+-----+-----+-----+
[5 rows x 5 columns]
```

```
query.head(5)
```

```
+-----+-----+-----+-----+
| street_number | address_1 | suburb | state | postcode |
+-----+-----+-----+-----+
|      31      |          | reseevoir | qld  | 5265   |
|     329      |          | smithfield plains | vic  | 5044   |
|      37      | kelleway avenue | burwooast | nsw  | 2770   |
|      15      | mawalan street | kallangur | nss  | 2506   |
|     380      | davisktrdet | boyne ilsand | nss  | 6059   |
+-----+-----+-----+-----+
[5 rows x 5 columns]
```

Basic usage

The record linker model is somewhat smart about what feature transformations are needed to accommodate the specified distance (although note that the street number and postcode fields had to be forced to strings when the datasets were imported). For our first pass, we'll use Jaccard distance, which implicitly concatenates all features (as strings), then converts them to character n-grams (a.k.a. shingles). In our experience, this is a good strategy for obtaining reasonable first results from address features.

```
model = gl.record_linker.create(refs, features=address_features,
                                 distance='jaccard')
model.summary()
```

```

Class : RecordLinker

Schema
-----
Number of examples : 3000
Number of feature columns : 5
Number of distance components : 1
Method : brute_force

Training
-----
Total training time (seconds) : 2.7403

```

Results are obtained with the model's `link` method, which matches a new set of queries to the reference data passed in above to the `create` function. For our first pass, we set the `radius` parameter to 0.5, which means that matches must share at least roughly 50% of the information contained in both the reference and query records.

```

matches = model.link(query, k=None, radius=0.5)
matches.head(5)

```

query_label	reference_label	distance	rank
1	2438	0.41935483871	1
1	533	0.5	2
2	688	0.192307692308	1
3	2947	0.045454545454545	1
5	1705	0.047619047619	1

[5 rows x 4 columns]

The results mean that the address in `query` row 1 match the address in `refs` row number 2438, although the Jaccard distance is relatively high at 0.42. Inspecting these records manually we see this is in fact *not* a good match.

```

print query[1], '\n'
print refs[2438]

```

```

{'suburb': 'smithfield plains', 'state': 'vic', 'address_1': '',
 'street_number': '329', 'postcode': '5044'}

{'suburb': 'smithfield plains', 'state': 'vic', 'address_1': 'sculptor street',
 'street_number': '16', 'postcode': '5044'}

```

On the other hand, the match between query number 3 and reference number 2947 has a distance of 0.045, indicating these two records are far more similar. By pulling these records we confirm this to be the case.

```
print query[3], '\n'
print refs[2947]
```

```
{'suburb': 'kallangur', 'state': 'nss', 'address_1': 'mawalan street',
'street_number': '15', 'postcode': '2506'}
```

```
{'suburb': 'kallangur', 'state': 'nsw', 'address_1': 'mawalan street',
'street_number': '12', 'postcode': '2506'}
```

Defining a composite distance

Unfortunately, these records are **still** not a true match because the street numbers are different (in a way that is not likely to be a typo). Ideally we would like street number differences to be weighted heavily in our distance function, while still allowing for typos and misspellings in the street and city names. To do this we can build a **composite distance** function.

A **composite distance** is simply a weighted sum of standard distance functions, specified as a list. Each element of the list contains three things:

1. a list or tuple of feature names,
2. a standard distance name, and
3. a multiplier for the standard distance.

Please see the documentation for the [GraphLab Create distances module](#) for more on composite distances.

In this case we'll use Levenshtein distance to measure the dissimilarity in street number, in addition to our existing Jaccard distance measured over all of the address features. Both of these components will be given equal weight. In the summary of the created model, we see the number of distance components is now two---Levenshtein and Jaccard distances---instead of one in our first model.

```
address_dist = [
    [['street_number'], 'levenshtein', 1],
    [address_features, 'jaccard', 1]
]

model = gl.record_linker.create(refs, distance=address_dist)
model.summary()
```

```

Class : RecordLinker

Schema
-----
Number of examples : 3000
Number of feature columns : 5
Number of distance components : 2
Method : brute_force

Training
-----
Total training time (seconds) : 0.4789

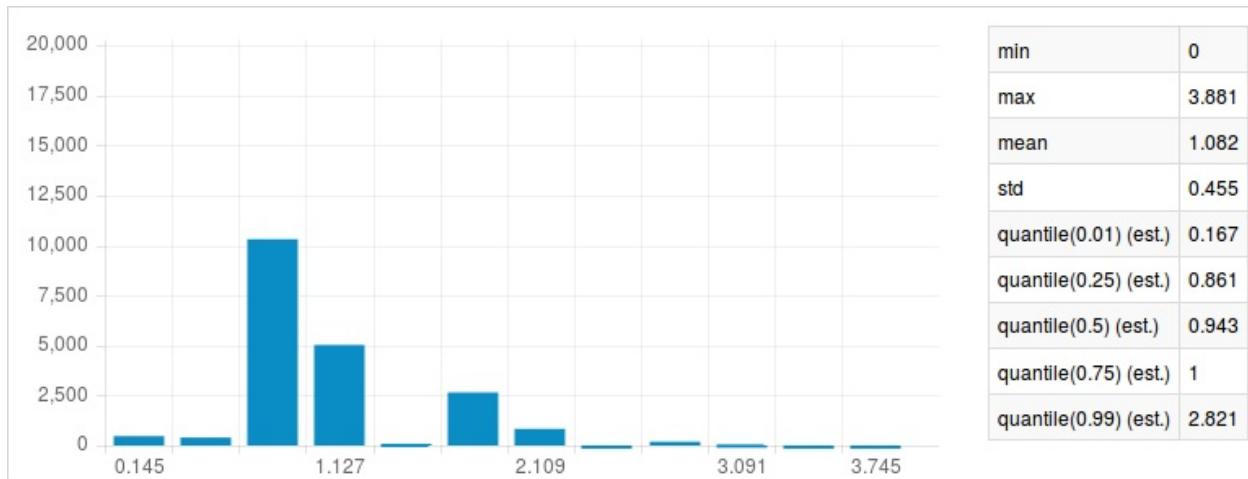
```

One tricky aspect of using a composite distance is figuring out the best threshold for match quality. A simple way to do this is to first return a relatively high number of matches for each query, then look at the distribution of distances for good thresholds using the `radius` parameter.

```

pre_match = model.link(query, k=10)
pre_match['distance'].show()

```



In this distribution we see a stark jump at 0.636 in the distribution of distances for the 10-nearest neighbors of every query (remember this is no longer simple Jaccard distance, but a sum of Jaccard and Levenshtein distances over different sets of features). In our final pass, we set the `k` parameter to `None`, but enforce this distance threshold with the `radius` parameter.

```

matches = model.link(query, k=None, radius=0.64)
matches.head(5)

```

query_label	reference_label	distance	rank
6	1266	0.333333333333	1
7	2377	0.208333333333	1
9	2804	0.575757575758	1
12	2208	0.181818181818	1
13	1346	0.111111111111	1

[5 rows x 4 columns]

There are far fewer results now, but they are much more likely to be true matches than with our first model, even while allowing for typos in many of the address fields.

```
print query[6], '\n'
print refs[1266]
```

```
{'suburb': 'clayton south', 'state': '', 'address_1': 'ingham oace',
'street_number': '128', 'postcode': '7520'}

{'suburb': 'clayton south', 'state': 'nsw', 'address_1': 'ingham place',
'street_number': '128', 'postcode': '7052'}
```

References

- Febrl - A Freely Available Record Linkage System with a Graphical User Interface. Peter Christen. Proceedings of the ‘Australasian Workshop on Health Data and Knowledge Management’ (HDKM). Conferences in Research and Practice in Information Technology (CRPIT), vol. 80. Wollongong, Australia, January 2008.

Deduplication

The GraphLab Create **deduplication** tool ingests data in one or more SFrames and assigns an entity label to each row. Records with the same label likely correspond to the same real-world entity.

To illustrate usage of the deduplication tool, we use data about musical albums, downloaded originally from <http://hpi.de/de/naumann/projects/data-quality-and-cleansing/annealing-standard.html#c123255>. For this example, we have extracted a random sample of about 20% of the original data, and split it into four SFrames based on genre. The preprocessed data can be downloaded (and saved to your machine) from the Turi public datasets bucket with the following code. This download is about 7MB.

```
import os
import graphlab as gl
import graphlab.aggregate as agg

genres = ['rock', 'americana', 'classical', 'misc']
data = {}

for g in genres:
    if os.path.exists('{}_albums'.format(g)):
        print "Loading genre '{}' from local SFrame....".format(g)
        data[g] = gl.load_sframe('{}_albums'.format(g))
    else:
        print "Downloading genre '{}' from S3 bucket....".format(g)
        data[g] = gl.load_sframe(
            'https://static.turi.com/datasets/dedupe_albums/{}_albums'.format(g))
        data[g].save('{}_albums'.format(g))
```

As usual, our first step is to look at the data:

```
data['rock'].print_rows(5)
```

disc_id	freedbdiscid	artist_name	disc_title	
166	1075217	Various	Mega Hits'80 S-03	
719	33670401	Dead Can Dance	Sambatiki	
829	34061313	Alice Cooper	Anselmo Valencia Amphitheatre	
1810	51495699	Kasabian	Kasabian-Ulimate Version-	
2013	68222994	Various Artists	Fear Candy14	
genre_title	disc_released	disc_tracks	disc_seconds	disc_language
Rock	1994	17	4200	eng
Alternative	1999	1	453	
Hard Rock	2003	1	1980	
Rock	2005	19	4547	
Metal	2005	18	4352	eng
...

[23202 rows x 9 columns]

For this example, we define a distance that is a weighted sum of Euclidean distance on the album length (in seconds), weighted Jaccard distance on the artist name and album title, and Levenshtein distance on the genre. Note that if you pass a composite distance to the deduplication toolkit, there is no need to specify the 'features' parameter; the composite distance already defines the relevant features.

```
album_dist = [[('disc_seconds',), 'euclidean', 1],
              [(['artist_name', 'disc_title'), 'weighted_jaccard', 4],
               [(['genre_title',), 'levenshtein', 1]]]
```

Grouping the data

In any dataset larger than about 10,000 records, grouping the records into smaller blocks (also known as "blocking") is critical to avoid computing the distance between all pairs of records.

In this example, we group on the number of tracks on each album ("disc_tracks"), which means that the toolkit will first split the data into groups each of whose members have the same number of tracks, then look for approximate matches only within each group. As of GraphLab Create v1.4, grouping features are specified with the `grouping_features` parameter; previous versions used the standard distance "exact" for this purpose, but this flag is no longer enabled.

Feature engineering

In the `nearest_neighbors` toolkit, the `weighted_jaccard` distance applies only to dictionary-type features, but in our `album_dist` we indicated that we want to apply it to two string-type features ('`artist_name`' and '`disc_title`'). The deduplication toolkit does several feature engineering steps automatically so you have less work to do manipulating the data and defining the composite distance. In particular:

- String features are cleaned by removing punctuation and extra white space, and converting all characters to lower case.
- Strings are converted to dictionaries with 3-character shingling when used with dictionary-based distances (`jaccard`, `weighted_jaccard`, `cosine`, and `dot_product`).
- String features specified for a single distance component are concatenated, separated by a space.
- Missing values are imputed. Missing strings are imputed to be "", while missing numeric values are imputed to be the mean value for the appropriate feature within the exact match group (see previous section). Note that records with missing values in the "exact" match features are ignored in model training and assigned entity label "None".

The feature engineering that occurs within the deduplication toolkit does not alter the input data in any way.

Choosing a model

Currently the deduplication toolkit has only one model: "`nearest_neighbor_deduplication`", which labels a pair of records as duplicate if one record is a close neighbor of the other. To resolve the question of *transitive closure*--A and B are duplicates, B and C are duplicates, but A and C are not--this model constructs a similarity graph and finds the connected components. Each connected component corresponds to an entity in the final output.

In addition to the data and the distance function, the nearest neighbor deduplication model takes two parameters. If `k` is specified, only the closest `k` neighbors for a record are considered duplicates, while the `radius` parameter indicates the maximum distance that a neighbor can be from a record to still be considered a duplicate. The most typical usage leaves `k` unspecified, and uses a `radius` that makes sense for the problem and the distance function.

```
m = gl.nearest_neighbor_deduplication.create(data, row_label='disc_id',
                                              grouping_features=['disc_tracks'],
                                              distance=album_dist,
                                              k=None, radius=3)
```

If two datasets are known to have records that match one-to-one, then setting `k=2` can be very useful. The `k` parameter is also useful to get preliminary results when we have no prior intuition about the problem or the distance function. In addition, the top level `deduplication.create` method hides the parameters, in the expectation that future versions of the toolkit will automatically choose the best modeling solution.

```
m2 = gl.deduplication.create(data, row_label='disc_id',
                               grouping_features=['disc_tracks'],
                               distance=album_dist)
```

Returning to our original model, the usual GraphLab Create toolkit functions give us information about model training and access to the output entity labels.

```
m.summary()
```

```
Class : NearestNeighborDeduplication

Schema
-----
Number of input datasets      : 4
Number of feature columns     : 4
Number of neighbors per point (k) : None
Max distance to a neighbor (radius) : 3
Number of entities           : 129632
Total training time (seconds)   : 268.9886

Training
-----
Total training time (seconds)   : 268.9886

Accessible fields
entities                  :
                           : Consolidated input records plus entity labels.
```

The model's `entities` attribute contains the deduplication results. All input data rows are appended into a single SFrame, and the column `__entity` indicates which records correspond to the same entity. Because we specified the datasets in a dictionary, the keys of that dictionary are used to identify which SFrame each record comes from; if the data were passed as a list this would be the index of the list.

Aggregating records

The entity labels are not very interesting by themselves; the deduplication problem typically involves a final step of aggregating records to produce a final clean dataset with one record per entity. This can be done straightforwardly with the SFrame groupby-aggregate tool.

In this example we define an aggregator that returns the number of records belonging to the entity, the mean length of album in seconds, the shortest album title and the compilation of all constituent genre and artist names.

```
# add disc title length in number of characters
entities = m['entities']
entities['title_length'] = entities['disc_title'].apply(lambda x: len(x))

# define the aggregation scheme and do the aggregation
album_aggregator = {
    'num_records': agg.COUNT,
    'disc_seconds': agg.MEAN('disc_seconds'),
    'genres': agg.CONCAT('genre_title'),
    'artist_name': agg.CONCAT('artist_name'),
    'title': agg.ARGIN('title_length', 'disc_title')
}

sf_clean = entities.groupby('__entity', album_aggregator)

# find the dupe IDs
entity_counts = m['entities'].groupby('__entity', agg.COUNT)
dupe_entities = entity_counts[entity_counts['Count'] > 1]['__entity']

# print the results for the dupes
dupes = sf_clean.filter_by(dupe_entities, '__entity')
dupes.print_rows(10, max_row_width=100, max_column_width=50)
```

__entity	genres	num_records	disc_seconds
76316	[Rock, Rock]	2	3154.5
16727	[Speech, Speech]	2	4770.0
73607	[Rock, Rock]	2	3498.0
17856	[Jazz, Jazz]	2	2817.0
25833	[Synthpop, Synthpop]	2	4202.0
16473	[Country, Country]	2	2055.0
76742	[Native American, Native American]	2	4217.0
80763	[Country, Country]	2	4520.5
26116	[Rock, Rock]	2	4640.0
43964	[Jazz, Jazz]	2	3357.0
artist_name	title		
[Cheap Trick, Cheap Trick]	Greatest Hits		
[Neville Jason, Neville Jason]	The Lives Of The Great Artists		
[Monster Magnet, Monster Magnet]	4-Way Diablo		
[Patricia Barber, Patricia Barber]	Split		
[Various, Various]	Atmospheric Synthesizer Spectacular Vol.2		
[Various, Various]	The Power Of Country		
[Paul Ortega, A. Paul Ortega-Two Worlds]	Three Worlds		
[Marvin Rainwater, Marvin Rainwater]	Classic Recordings Disc1		
[Various, Various]	Rules Of Rock		
[Woong San, Woomg San]	Close Your Eyes		

[321 rows x 6 columns]

Autotagger

The GraphLab Create **autotagger** tool matches unstructured text queries to a reference set of strings, a.k.a *tags*, which are known beforehand. Adding tags to unstructured text gives readers a quick intuition about the content of the text as well as anchors for quicker navigation and statistical summaries. Autotagging is closely related to the task of *searching* for user-specified queries in unstructured text, but it differs in that autotagging is typically done with a fixed set of tags, and treats the unstructured documents as queries, rather than the tags.

In this chapter we autotag posts from [CrossValidated](#), the statistics section of the Stack Exchange network. Questions posted on this forum are typically annotated with tags by the authors but responses are not, making it more difficult to quickly scan responses for the most useful information. The raw data is available from the [Stack Exchange data dump](#). For convenience we provide a preprocessed subsample (7.8MB) in the public Turi datasets bucket on Amazon S3, which is downloaded and saved locally with the first code snippet below.

For reference tags we use a lightly-curated [list of statistics topics from Wikipedia](#). The preprocessed list is also available in the static.turi.com/datasets S3 bucket.

```
import os
import graphlab as gl

## Load unstructured text data
if os.path.exists('stats_overflow_clean'):
    posts = gl.SFrame('stats_overflow_clean')
else:
    posts = gl.SFrame('https://static.turi.com/datasets/stats_overflow_clean')
posts.save('stats_overflow_clean')

## Load reference set of statistics topics
if os.path.exists('statistics_topics.csv'):
    topics = gl.SFrame.read_csv('statistics_topics.csv', header=False, delimiter='\n')
else:
    topics = gl.SFrame.read_csv(
        'https://static.turi.com/datasets/tag_lists/statistics_topics.csv',
        header=False, delimiter='\n')
topics.save('statistics_topics', format='csv')

topics.rename({'X1': 'topic'})
```

Here's a quick peek at the data in both the reference tags and the CrossValidated posts. There are 2,737 topics and 11,077 posts, comprising both original questions and response. The 'PostTypeID' column indicates whether a row corresponds to a question or a response, and it's clear that the responses (PostTypeID == 2) have neither tags nor titles.

```
topics.print_rows(5)
```

```
+-----+
|      topic      |
+-----+
| A priori probability   |
| Abductive reasoning    |
| Absolute deviation     |
| Absolute risk reduction|
| Absorbing Markov chain |
| ...                   |
+-----+
[2737 rows x 1 columns]
```

```
posts.print_rows(5, max_row_width=100)
```

```
+-----+-----+-----+
| AnswerCount | Body          | ClosedDate | CommentCount |
+-----+-----+-----+
| None       | Assuming you meant a binom... | None       | 0           |
| None       | This is because you are fi... | None       | 0           |
| None       | I think I agree, drag/drop... | None       | 0           |
| None       | Similar to Weka, you may a... | None       | 0           |
| None       | Scottchi and Peter Flom ha... | None       | 3           |
+-----+-----+-----+
+-----+-----+-----+-----+-----+
| CreationDate | FavoriteCount | PostTypeID | Score | Tags | Title |
+-----+-----+-----+-----+-----+
| 2014-06-01 00:03:48+00:00 | None       | 2          | 3     | None | None |
| 2014-06-01 00:09:09+00:00 | None       | 2          | 1     | None | None |
| 2014-06-01 01:26:06+00:00 | None       | 2          | 1     | None | None |
| 2014-06-01 01:29:40+00:00 | None       | 2          | 0     | None | None |
| 2014-06-01 01:32:17+00:00 | None       | 2          | 5     | None | None |
+-----+-----+-----+-----+-----+
[11077 rows x 10 columns]
```

There is currently only one autotagger model, accessible through the `graphlab.autotagger.create` call. This method takes the reference tag data and returns a `NearestNeighborAutoTagger` model, which can then be queried with the unstructured text

data. Under the hood, the model cleans input strings (in both the tag and query datasets), generates unigrams, bigrams, and 4-character shingles, and computes the distance between tags and queries with weighted Jaccard distance.

```
m = gl.autotagger.create(topics, verbose=False)
m.summary()
```

```
Class : NearestNeighborAutoTagger

Settings
-----
Number of examples      : 2732
Number of feature columns : 3
Total training time (seconds) : 0.0514
```

There are two key parameters when querying the model: `k`, which indicates the maximum number of tags to return for each query, and `similarity_threshold`, which indicates the minimum similarity from a query document to the tag. A typical pattern is to get preliminary results by setting `k` to 5 and leaving `similarity_threshold` unspecified, then run `tag` again using the `similarity_threshold` parameter for finely-tuned results.

The query documents must be a single column of the query SFrame, so we first concatenate the CrossValidate post titles and bodies.

```
posts['all_text'] = posts['Title'] + ' ' + posts['Body']
tags = m.tag(posts, query_name='all_text', k=5, similarity_threshold=0.1,
             verbose=True)
tags.print_rows(10, max_row_width=100, max_column_width=50)
```

```

PROGRESS: Starting pairwise querying.
PROGRESS: +-----+-----+-----+
PROGRESS: | Query points | # Pairs | % Complete. | Elapsed Time |
PROGRESS: +-----+-----+-----+
PROGRESS: | 0 | 692 | 0.00228667 | 26.525ms |
PROGRESS: | 4844 | 1.3e+07 | 43.7346 | 1.02s |
PROGRESS: | 9692 | 2.6e+07 | 87.4989 | 2.03s |
PROGRESS: | Done | | 100 | 2.29s |
PROGRESS: +-----+-----+-----+
+-----+
| all_text_id | all_text |
+-----+
| 13 | neural network output layer for binary classif... |
| 13 | neural network output layer for binary classif... |
| 13 | neural network output layer for binary classif... |
| 13 | neural network output layer for binary classif... |
| 13 | neural network output layer for binary classif... |
| 37 | Negative predictions for binomial predictions ... |
| 55 | Estimating entropy of multidimensional variabl... |
| 80 | Does the sequence satisfy WLLN? Could you help... |
| 80 | Does the sequence satisfy WLLN? Could you help... |
| 80 | Does the sequence satisfy WLLN? Could you help... |
+-----+
+-----+
| topic | score |
+-----+
| Binary classification | 0.15503875969 |
| Artificial neural network | 0.107913669065 |
| One-class classification | 0.101449275362 |
| Neural network | 0.100775193798 |
| Multiclass classification | 0.10071942446 |
| Negative predictive value | 0.104712041885 |
| Dimension reduction | 0.101123595506 |
| Law of large numbers | 0.197916666667 |
| Independent and identically distributed random... | 0.186046511628 |
| Convergence of random variables | 0.177570093458 |
+-----+
[1356 rows x 4 columns]

```

Note that the `score` column in the tags output is a *similarity* score, unlike the `radius` parameter in many other tools in GraphLab Create. The similarity in the autotagger is simply 1 minus the weighted jaccard distance, so setting the `similarity_threshold` to 0.1, for example, requires that all output query-tag matches have a weighted jaccard distance of no more than 0.9. In these particular results we see that that a post that appears to be about "binary classification" indeed receives this topic as its top match, while the post about the weak law of large numbers (WLLN) is appropriately tagged with "Law of large numbers."

Post-processing of the autotagger toolkit generally involves straightforward SFrame operations. For example, suppose we want to compare our model-generated tags to the human-generated ones attached to original question posts. To do this we first *filter* the unstructured dataset of posts by post type, then *unstack* the tags output to get a list of tags for each query in a single row, and finally *join* the original posts to the tags.

```
tags.rename({'all_text_id': 'id'})  
tags = tags[['id', 'topic']].unstack('topic', new_column_name='topics')  
  
posts = posts.add_row_number('id')  
tags = tags.join(posts[['Body', 'Title', 'Tags', 'id']], on='id', how='left')  
  
print tags[0]
```

```
{'Body': "I'm using a neural network for a binary classification problem. Is it  
better to have one neuron in the output layer or to use two, i.e. one for each  
class? ",  
'Tags': '',  
'Title': 'neural network output layer for binary classification',  
'id': 13,  
'topics': ['Binary classification',  
'One-class classification',  
'Artificial neural network']]}
```

In this particular example, both the human and the autotagger used a neural networks tag, while the human also attached a more general "machine learning" tag, and the autotagger included a "binary classification" tag.

Similarity Search

Similarity search is the task of matching complex data objects like images and documents. Like [nearest neighbors search](#), similarity search matches query items to a fixed reference set, but similarity search allows the input data to be in a raw form, such as images, documents, or combinations of the two. Similarity search is also similar to [autotagging](#), but for autotagging the reference tags are always entered in the form of a table; similarity search is more general.

The Similarity Search toolkit encapsulates in a single framework the entire process of type detection, feature engineering, and search. For quick prototyping, the toolkit intelligently detects the type of input data and selects an appropriate pre-trained model for converting complex data objects into numeric features. For more customized usage, users can inspect and modify each of the individual steps.

The initial version of the similarity search toolkit focuses exclusively on matching query images to reference images. In future versions, the toolkit will also handle documents and combinations of images and documents. *Feedback is most welcome: help us make this tool as useful as possible!*

MNIST Digits Example

To illustrate usage of the similarity search toolkit, we use a small subset of the [MNIST handwritten digits image dataset](#), which is downloaded from the public Turi datasets bucket on Amazon S3. The download is about 1.5 MB.

```
import os
import graphlab as gl

## Download and/or load image data
if os.path.exists('mnist_train6k'):
    mnist = gl.SFrame('mnist_train6k')
else:
    mnist = gl.SFrame('https://static.turi.com/datasets/mnist/sframe/train6k')
    mnist.save('mnist_train6k')
```

The dataset is very simple: each row contains a GraphLab Create Image object and the number shown in the image.

```
mnist.print_rows(5)
```

```
+-----+-----+
| label |      image      |
+-----+-----+
| 5    | Height: 28 Width: 28 |
| 8    | Height: 28 Width: 28 |
| 1    | Height: 28 Width: 28 |
| 4    | Height: 28 Width: 28 |
| 2    | Height: 28 Width: 28 |
+-----+
[6000 rows x 2 columns]
```

The simplest way to construct a similarity search model is to let the toolkit use a default neural net model for converting the images into numeric vectors. The only required arguments are the reference dataset and the name of the column containing images.

```
search_model = gl.similarity_search.create(mnist, features='image')
```

While this model is very simple to code, it can be quite slow to create. The model downloads a 500MB pre-trained [ImageNet neural net classifier model](#) from the public Turi datasets Amazon S3 bucket, and then modifies the images to work with the model.

A more sophisticated way to create a similarity search model is to construct a new neural net classifier tailored to the data of interest, then pass this model explicitly to the similarity search toolkit.

```
nn_model = gl.neuralnet_classifier.create(mnist, target='label')
search_model2 = gl.similarity_search.create(mnist, features='image',
                                             feature_model=nn_model)
search_model2.summary()
```

```
Class : SimilaritySearchModel

Schema
-----
Number of reference examples : 6000

Training
-----
Method : lsh
Total training time (seconds) : 3.4045
```

The model summary shows that the default search method is `lsh`, short for [locality-sensitive hashing](#). In the model creation step, all reference images are assigned to a bucket by passing them through a set of hash functions designed to approximate cosine distance. A

query image is then passed through the same hash functions, assigning it to one of the original buckets. The reference images in that bucket are candidate neighbors and are then compared to the query and ranked by computing the exact cosine distance.

The LSH method makes queries very fast, but yields a set of similar items that is **approximate**, in that some results may not very similar, and some similar items may not be returned. For exact results, the `method` parameter can be set to `brute_force` when the model is created. For this example, we stick with locality-sensitive hashing, and find the three most similar items for the first four images in our training set.

```
sim_items = search_model2.search(mnist[:4], k=3)
sim_items.print_rows(12)
```

query_label	reference_label	distance	rank
0	0	-2.22044604925e-16	1
0	2308	6.42280780891e-05	2
0	2762	0.000111448855981	3
1	1	0.0	1
1	5796	0.000101432998189	2
1	859	0.000116476316971	3
2	2	1.11022302463e-16	1
2	918	0.00013260851351	2
2	5907	0.000174541914201	3
3	3	0.0	1
3	4426	0.00010230738112	2
3	5344	0.000183852540399	3

[12 rows x 4 columns]

Our first sanity check is that the most similar item to each query is itself - success! To check the rest of the results, we join the original labels and images to the similarity search results.

```
mnist = mnist.add_row_number('id')
sim_items = sim_items.join(mnist, on={'reference_label': 'id'}, how='left')
sim_items = sim_items.sort(['query_label', 'reference_label'])
sim_items.print_rows(12, max_row_width=120)
```

query_label	reference_label	distance	rank	label	image
0	0	-2.22044604925e-16	1	5	Height: 28 Width: 2
0	2308	6.42280780891e-05	2	7	Height: 28 Width: 2
0	2762	0.000111448855981	3	9	Height: 28 Width: 2
1	1	0.0	1	8	Height: 28 Width: 2
1	859	0.000116476316971	3	0	Height: 28 Width: 2
1	5796	0.000101432998189	2	8	Height: 28 Width: 2
2	2	1.11022302463e-16	1	1	Height: 28 Width: 2
2	918	0.00013260851351	2	1	Height: 28 Width: 2
2	5907	0.000174541914201	3	6	Height: 28 Width: 2
3	3	0.0	1	4	Height: 28 Width: 2
3	4426	0.00010230738112	2	5	Height: 28 Width: 2
3	5344	0.000183852540399	3	4	Height: 28 Width: 2

[12 rows x 6 columns]



These results appear better than random, but there is substantial room for improvement. The first option is to do similarity search with the `brute_force` method, but another likely improvement would be a more powerful neural network model for the feature engineering step.

Lead Scoring

Prioritizing new leads is critical for sales and marketing teams. Traditionally, this is done by giving each open sales account a point value, and adding a predetermined number points each time an account interacts with your business in some way. The accounts with the point totals are declared to be the highest priority.

The **Lead Scoring toolkit** in GraphLab Create (GLC) is more intelligent. By using historical data to learn the relationship between account behavior and the successful conversions, this tool can predict future conversions much more accurately. In turn, this allows sales and marketing resources to be allocated more efficiently to accounts or market segments with the most value, saving time and money.

There are two tasks that people typically want to accomplish with a lead scoring model:

1. predict whether specific open accounts will successfully convert to a sale
2. segment the market into accounts with similar features and conversion probabilities.

The GLC Lead Scoring toolkit accomplishes both of these tasks simultaneously with a combination of advanced feature engineering, a gradient boosted trees model to predict conversions, and a decision tree model to construct market segments.

Basic Usage

The primary unit of interest in lead scoring is the **sales account**. This is typically a company or an individual user. Accounts fall into one of three buckets, depending on their **conversion status**:

1. **Successful conversions**. Accounts which have purchased your product or service.
2. **Failures**. Accounts which you have determined will not purchase.
3. **Open**. Accounts which have not yet decided whether they will buy or not.

The lead scoring toolkit uses the accounts in the first two buckets (together called **training accounts**) to learn the relationship between account features and the conversion outcome, then predicts the outcome for the accounts that remain open.

The simplest possible usage of the lead scoring toolkit requires only this information. To illustrate, we use data from the [AirBnB New User Bookings challenge on Kaggle](#), although we will change the task. Let's imagine we are an online travel agency, and our goal is to

convince users to book trips outside of the United States. That is, successfully converted accounts are users with non-US bookings, failed accounts are US bookings, and the competition's "test users" are our open accounts.

```
import graphlab as gl
users = gl.SFrame('airbnb_users.sfr')
users.print_rows(5)
```

id	date_account_created	date_first_booking	gender	age
4ft3gnwmtx	2010-09-28 00:00:00	2010-08-02 00:00:00	FEMALE	56.0
bjjt8pjhuk	2011-12-05 00:00:00	2012-09-08 00:00:00	FEMALE	42.0
lsw9q7uk0j	2010-01-02 00:00:00	2010-01-05 00:00:00	FEMALE	46.0
0d01nltbtrs	2010-01-03 00:00:00	2010-01-13 00:00:00	FEMALE	47.0
a1vcnhxeij	2010-01-04 00:00:00	2010-07-29 00:00:00	FEMALE	50.0

signup_method	signup_app	first_device_type	first_browser	conversion
basic	Web	Windows Desktop	IE	-1
facebook	Web	Mac Desktop	Firefox	1
basic	Web	Mac Desktop	Safari	-1
basic	Web	Mac Desktop	Safari	-1
basic	Web	Mac Desktop	Safari	-1

[73067 rows x 10 columns]

Note the values in the "conversion" column: **-1 indicates a failed account, +1 indicates a successful conversion, and 0 indicates an account that is open and needs to be scored.**

The `account_schema` parameter lets us specify which columns of the data should be used as conversion status, ID, features, etc. For the most basic usage, the only required entries are `account_id` and `conversion_status`, but we will manually specify the features we want to use as well.

```
user_schema = {
    'conversion_status': 'conversion',
    'account_id': 'id',
    'features': ['gender', 'age', 'signup_method', 'signup_app',
                 'first_device_type', 'first_browser']}

model = gl.lead_scoring.create(users, user_schema)
print(model)
```

```

Class : LeadScoringModel

Model schema
-----
Number of accounts : 73067
Number of interactions : 0
Number of account features : 6
Number of interaction features : 0
Verbose : True

Training summary
-----
Number of training accounts : 52321
Number of open accounts : 20746
Number of successful conversions : 15454
Number of explicit failures : 36867
Number of implicit failures : 0
Number of final features : 6
Total training time (seconds) : 5.1916

Accessible fields
-----
open_account_scores : Lead scores, segment IDs, and final
                      features for open accounts.
training_account_scores : Lead scores, segment IDs, and final
                           features for training accounts.
segment_descriptions : Statistics about market segments of
                        training (i.e. closed) accounts.
scoring_model : Underlying GBT model for predicting
                  account conversion.
segmentation_model : A trained decision tree to create
                     account segments.
account_schema : Schema for the 'accounts' input.
interaction_schema : Schema for the 'interactions'SFrame
                     (if provided).

```

The model summary indicates the breakdown of our accounts by status: of the total 73,067 users, 20,746 are *open* and need to be scored, 36,867 have *failed*, and 15,454 have *successfully converted*.

The summary also shows the fields that are accessible in a trained lead scoring model. The three most important outputs are `open_account_scores` , `training_account_scores` , and `segment_descriptions` .

`model.open_account_scores` contains the model's lead score ("conversion_prob") and market segment assignment ("segment_id") for each open account, as well as the features used to obtain those results. Use the `SFrame.topk` function to see which open accounts are mostly likely to convert.

```
print(model.open_account_scores.topk('conversion_prob', 3))
```

account_id	gender	age	signup_method	signup_app	first_device_type
9zec42xc6n	FEMALE	65.0	basic	iOS	iPad
3ea6j7awkf	FEMALE	60.0	basic	iOS	iPad
wbc3f9jzeo	FEMALE	63.0	basic	iOS	iPad

first_browser	conversion_prob	segment_id
Mobile Safari	0.7325757145881653	13
Mobile Safari	0.7152470946311951	13
Mobile Safari	0.7152470946311951	13

[3 rows x 9 columns]

`model.training_account_scores` contains the estimated conversion probability (i.e. fitted value) and market segment assignment for training accounts, i.e known failures and successful conversions, as well as the final features used by the model.

```
print(model.training_account_scores.head(3))
```

account_id	conversion_status	gender	age	signup_method	signup_app
4ft3gnwmtx	-1	FEMALE	56.0	basic	web
bjjt8pjhuk	1	FEMALE	42.0	facebook	Web
lsw9q7uk0j	-1	FEMALE	46.0	basic	Web

first_device_type	first_browser	conversion_prob	segment_id
Windows Desktop	IE	0.30257749557495117	13
Mac Desktop	Firefox	0.3005772829055786	12
Mac Desktop	Safari	0.2826768159866333	12

[3 rows x 10 columns]

`model.segment_descriptions` includes the definitions of the market segments in terms of features and summary statistics about each segment, based on the training accounts in each segment. The segment IDs match the segment assignments in the previous two tables (although note that segment IDs they are *not* consecutive integers).

```
description_cols = ['segment_id', 'mean_conversion_prob',
                    'num_training_accounts', 'segment_features']
model.segment_descriptions[description_cols].print_rows(max_column_width=20)
```

segment_id	mean_conversion_prob	num_training_acc...	segment_features
10	0.4157824154170054	1590	[age < 23.5, fir...
8	0.38625910013914094	100	[age < 23.5, fir...
14	0.35017348461894743	3948	[age >= 47.5, ge...
13	0.31760550252200537	4590	[age >= 47.5, ge...
7	0.3093886854123114	946	[age < 23.5, fir...
12	0.29379484518361165	38936	[age in [23.5, 4...
9	0.2894578479230403	60	[age < 23.5, fir...
11	0.2241769344626004	2151	[age in [23.5, 4...

[8 rows x 4 columns]

The feature descriptions in this table are *conjunctions*; all of the conditions must be true for an account to belong to a given segment. For example, the first row of the current model's segment descriptions is:

```
model.segment_descriptions[description_cols][0]
```

```
{'mean_conversion_prob': 0.4157824154170054,
 'num_training_accounts': 1590,
 'segment_features': ['age < 23.5',
 'first_device_type = Mac Desktop',
 'signup_app = Web'],
 'segment_id': 10}
```

This means that segment 10 includes accounts where the user's age is less than 23.5, the user first used a Mac on the site, and signed up for the site on the web. Of the 52,321 training accounts, 1,590 belong to this segment, and the average estimated conversion probability for these training accounts is 41.6%.

Accessing the scoring and segmentation models

The GLC Lead Scoring tool uses a [gradient boosted trees classifier](#) to make conversion predictions for the open accounts and a [decision tree regression](#) (trained with the scoring model's fitted values as the target) to create the market segments. These internal models are available if we want to see more details about model training.

Furthermore, unlisted keyword arguments provided to the lead scoring create function are passed directly to gradient boosted trees classifier. For example, to change the maximum number of iterations in the classifier:

```
model2 = gl.lead_scoring.create(users, user_schema, max_iterations=4)
print(model2.scoring_model.num_trees)
```

4

Changing the number of market segments without re-training

By default the lead scoring model creates a maximum of 8 market segments. Because constructing features can take a lot of time for large datasets, the number of market segments can be changed without re-training from scratch.

```
model3 = model2.resize_segmentation_model(max_segments=20)

cols = ['segment_id', 'mean_conversion_prob', 'num_training_accounts']
model3.segment_descriptions[cols].print_rows(20)
```

segment_id	mean_conversion_prob	num_training_accounts
20	0.4765985885019464	466
15	0.4483680009841919	80
19	0.4243511390354903	540
21	0.4238594039745135	591
18	0.40929717818895994	78
28	0.401472113483039	1316
16	0.3736430677500637	132
22	0.3636468232802626	53
27	0.3619968469430385	8327
30	0.34992563210981964	9680
17	0.3479588490235742	756
26	0.3404770083202581	18524
25	0.3265099958305023	9814
29	0.3017061043331641	262
23	0.2956991525199296	1178
24	0.2512635290622711	524

[16 rows x 3 columns]

Incorporating interaction data

Often the interactions between accounts and business assets (e.g. websites, products, email campaigns) contain more information about each account's intent to advance to the next stage of the sales process. The GLC Lead Scoring tool accepts this type of data through the `interactions` parameter.

In addition to a `users` table, the AirBnB Kaggle challenge provides "session" data that describes every interaction between a user and the site, in terms of the action done by the user, the time elapsed for the action, and the user's device type. For this demo, I have generated random timestamps and converted the sessions table to a `TimeSeries` object (required for the GLC lead scoring tool).

```
sessions = gl.load_timeseries('airbnb_sessions.sfr')
sessions.print_rows(3)
```

timestamp	user_id	action	action_type
2013-12-30 00:09:05.777678	jrqykh9y8x	show	None
2013-12-30 00:51:37.515840	jrqykh9y8x	ajax_refresh_subtotal	click
2013-12-30 00:55:48.004325	mrvjvk6ycy	confirm_email	click

action_detail	device_type	secs_elapsed
None	Mac Desktop	106.0
change_trip_characteristics	Mac Desktop	408.0
confirm_email_link	Mac Desktop	42711.0

[3267114 rows x 7 columns]

The lead scoring tool uses interaction data to construct additional account-level features. To make sure this happens cleanly, additional information about each account must be provided.

- `open_date` : the date on which each account was opened.
- `decision_date` : the date on which training accounts either converted or were determined to be failures. Values in this column are ignored for the open accounts.

As with the accounts, the `interaction_schema` parameter is used to indicate columns in the interactions table that are relevant to the model. In this case, we specify that the account ID column is called "user_id", that the items of interest on the website are actually user "actions", and our additional interaction features are the device type used for the interaction and the length of the interaction in seconds.

```
user_schema.update({'decision_date': 'date_first_booking',
                    'open_date': 'date_account_created'})

session_schema = {'account_id': 'user_id',
                   'item': 'action',
                   'features': ['device_type', 'secs_elapsed']}
```

When interaction data is provided, we also must define a ***trial_duration***; in combination with an account's open date, this defines the time window for the interactions the model will use to construct features. In this example we use a 2-year trial period.

```
import datetime as dt

model = gl.lead_scoring.create(users, user_schema, sessions, session_schema,
                                trial_duration=dt.timedelta(days=365 * 2))
model.summary()
```

```

Class : LeadScoringModel

Model schema
-----
Number of accounts : 73067
Number of interactions : 3267114
Number of account features : 6
Number of interaction features : 2
Verbose : True

Training summary
-----
Number of training accounts : 52299
Number of open accounts : 20746
Number of successful conversions : 15448
Number of explicit failures : 36851
Number of implicit failures : 238
Number of final features : 8
Total training time (seconds) : 16.1299

Accessible fields
-----
open_account_scores : Lead scores, segment IDs, and final
                      features for open accounts.
training_account_scores : Lead scores, segment IDs, and final
                           features for training accounts.
segment_descriptions : Statistics about market segments of
                        training (i.e. closed) accounts.
scoring_model : Underlying GBT model for predicting
                  account conversion.
segmentation_model : A trained decision tree to create
                     account segments.
account_schema : Schema for the 'accounts' input.
interaction_schema : Schema for the 'interactions'SFrame
                     (if provided).

```

The names of interaction-based features created by the lead scoring tool are contained in the model's `open_account_scores` and `training_account_scores` outputs, along with the features we specified explicitly (as well as other fields specified in the account schema, plus the conversion probability and segment assignment). In this example, the model has created the 'num_events' and 'num_items' features.

```
model.open_account_scores.topk('conversion_prob', k=3).print_rows()
```

account_id	open_date	decision_date	gender	age	signup_method
ja20a5424q	2014-08-14 00:00:00	None	FEMALE	63.0	facebook
oacsjhuljj	2014-08-02 00:00:00	None	OTHER	19.0	basic
kbi9nrwao1	2014-07-22 00:00:00	None	MALE	105.0	basic
signup_app	first_device_type	first_browser	num_events		
iOS	iPad	Mobile Safari	0.1090717923876873		
Web	Windows Desktop	IE	0.048597775817815644		
Web	iPad	Mobile Safari	0.26315030503316156		
num_items	conversion_prob	segment_id			
0.024722939607875787	0.5543753504753113	13			
0.017152156170993756	0.5136026740074158	7			
0.06473216059639268	0.5080435276031494	13			

[3 rows x 13 columns]

Implicit failure accounts

Many sales and marketing teams do not explicitly label accounts that fail to convert; they record only successful conversions and open accounts. In this case the lead scoring tool can automatically determine ***implicit failure accounts***, which are accounts that have been open for so long their chances of converting to a sale are effectively nil. In order to find implicit failures, three key dates about the accounts need to be provided to the model:

- `open_date` : date on which the account opened. Applies to both training and open accounts.
- `decision_date` : date on which a training account's outcome was decided. This date must occur after the account open date, or the account is declared invalid. Decision dates for open accounts are ignored; by definition an open account has not been decided.
- `trial_duration` : length of the trial period. An open account that's still in its trial period is not declared an implicit failure, no matter how quickly the training accounts convert.

In the event that failures are not defined explicitly, and no implicit failures are found, the lead scoring toolkit will return an error and ask us to define failures manually. In our running illustration we currently have no implicit failures, but decreasing the trial period changes the situation.

```
model2 = gl.lead_scoring.create(users, user_schema, sessions, session_schema,
                                 trial_duration=dt.timedelta(days=365 * 1.9))

print(model2.num_implicit_failures)
```

```
8890
```

References and more information

- [GraphLab Create Lead Scoring API documentation](#)
- [AirBnB New User Bookings Kaggle competition](#)

Churn prediction

Churn prediction is the task of identifying whether users are likely to stop using a service, product, or website. With this toolkit, you can start with raw (or processed) usage metrics and accurately forecast the probability that a given customer will churn.

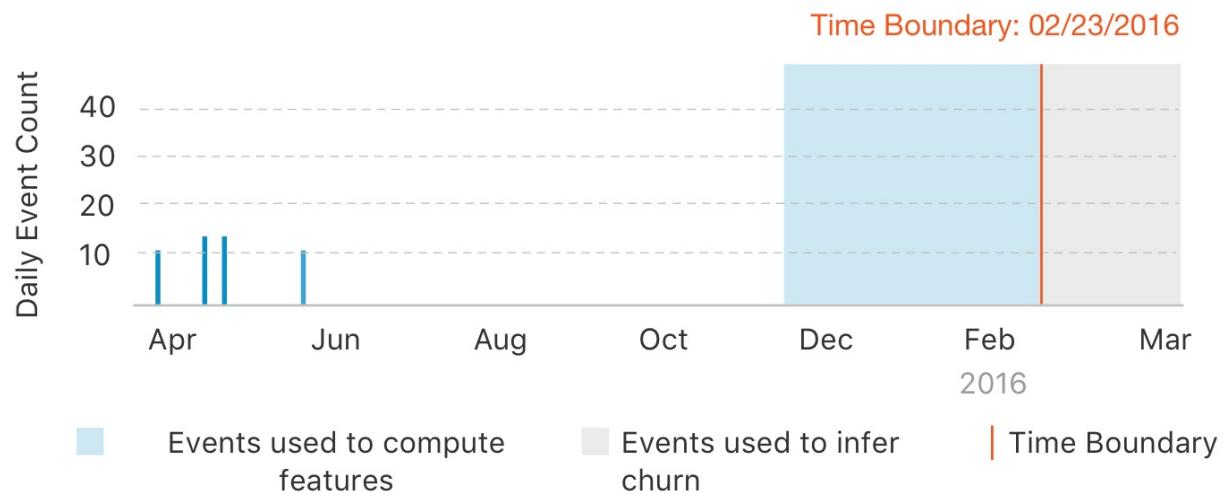
Note: Follow the steps in the [sample-churn-predictor](#) GitHub repo to get the code and data used in this chapter.

Introduction

A `churn predictor model` learns historical user behavior patterns to make an accurate forecast for the probability of no activity in the future (defined as churn).

How is churn defined?

Customer churn can be defined in many ways. In this toolkit, churn is defined to be **no activity** for a fixed period of time (called the `churn_period`). Using this definition, a user is said to have churned if there is **no activity** for a duration of time known as the `churn_period` (by default, this is set to 30 days). The following figure better illustrates this concept.



A churn forecast is always associated with a particular timestamp (at which the `churn_period` starts) known as a `time_boundary`. As an example, a user is said to have **churned** at the `time_boundary` Jan 2015 because the user did not have any activity for a `churn_period` duration of time (say 30 days) after Jan 2015.

Input Data

A churn prediction model can be trained on time-series of `observation_data`. The time-series must contain a column to represent `user_id` and at least one other column that can be treated as a feature column. The following example shows a typical dataset that can be consumed directly by the churn predictor toolkit.

InvoiceDate	CustomerID	Quantity
2010-12-01 08:26:00	17850	6
2010-12-01 08:26:00	17850	6
2010-12-01 08:26:00	17850	8
2010-12-01 08:26:00	17850	6
2010-12-01 08:26:00	17850	6
2010-12-01 08:26:00	17850	2
2010-12-01 08:26:00	17850	6
2010-12-01 08:28:00	17850	6
2010-12-01 08:28:00	17850	6
2010-12-01 08:34:00	13047	32
2010-12-01 08:34:00	13047	6
2010-12-01 08:34:00	13047	6
2010-12-01 08:34:00	13047	8
2010-12-01 08:34:00	13047	6
2010-12-01 08:34:00	13047	6
2010-12-01 08:34:00	13047	3
2010-12-01 08:34:00	13047	2
2010-12-01 08:34:00	13047	3
2010-12-01 08:34:00	13047	3
2010-12-01 08:34:00	13047	4

[532618 rows x 5 columns]

In the above dataset, let us assume that the last timestamp was October 1,

1. If the `churn_period` is set to 1 month, a churn forecast predicts the probability that a user will have no activity for a 1 month period after October 1, 2011.

Example

In this example, we will explore the task of predicting churn directly from customer activity logs. The following [dataset](#) contains transactions occurring between 01/12/2010 and 09/12/2011 for a UK-based and registered non-store online retail.

```
import graphlab as gl
import datetime

# Load a data set.
sf = gl.SFrame(
    'https://static.turi.com/datasets/churn-prediction/online_retail.csv')

# Convert InvoiceDate from string to a Python datetime.
import dateutil
from dateutil import parser
sf['InvoiceDate'] = sf['InvoiceDate'].apply(parser.parse)

# Convert the SFrame into TimeSeries with InvoiceDate as the index.
time_series = gl.TimeSeries(sf, 'InvoiceDate')

# Split the data using the special train, validation split.
train, valid = gl.churn_predictor.random_split(time_series,
                                               user_id='CustomerID', fraction=0.9)

# Define the period of in-activity that constitutes churn.
churn_period = datetime.timedelta(days = 30)

# Train a churn prediction model.
model = gl.churn_predictor.create(train, user_id='CustomerID',
                                   features = ['Quantity'],
                                   churn_period = churn_period)

# Making a churn forecast
predictions = model.predict(time_series)

# Evaluating the model
evaluation_time = datetime.datetime(2011, 9, 1)
predictions = model.predict(time_series, evaluation_time)

# Visualize the results
views = model.views.overview(time_series, evaluation_time)
views.show()
```

Learn more

The following sections provide more information about the churn prediction model:

- [Using a trained model](#)
- [Alternate input formats](#)
- [How it works](#)

Using a trained churn predictor model

In this chapter, we will explore how to use a trained churn predictor model. This includes the following:

- Making a churn forecast
- Evaluating the model
- Generating a churn report
- Explaining the predictions
- Accessing the model & underlying features
- Visualizing the model
- Saving & loading the model

Making predictions (a churn forecast)

The goal of a churn prediction model is to predict the probability that a user has no activity for a `churn_period` of time in the future. Hence, the output of this model is a forecast of what might happen **in the future**. The following example illustrates this concept.

```
predictions = model.predict(time_series)
```

CustomerID	probability
None	0.0661627277732
12346	0.67390537262
12347	0.760785758495
12348	0.62475168705
12349	0.372504591942
12350	0.67390537262
12352	0.201043695211
12353	0.821378648281
12354	0.737500548363
12355	0.699232280254

[4340 rows x 2 columns]

Note that you can also set a specific time at which the predictions must be made.

```
prediction_time = datetime.datetime(2011, 9, 1)
predictions = model.predict(time_series, time_boundary = prediction_time)
```

CustomerID	probability
None	0.0358778424561
12346	0.884932994843
12347	0.488953769207
12348	0.861040890217
12350	0.871586084366
12352	0.852140307426
12353	0.778547585011
12354	0.83321160078
12355	0.826072216034
12356	0.815514743328

[4373 rows x 2 columns]

During predictions, you may get a message that says `Not enough data to make predictions for 1005 user(s)`. There could be several reasons why predictions cannot be made for a set of users, they are:

- The user did not have any activity **before** the prediction time.
- The user did not have any activity for a period of time used during the feature engineering process (controlled by the parameter `lookback_periods`). The model looks at the usage history in the recent past to make a forecast. If no activity was present during this recent past, then predictions cannot be made. A prediction of `None` is returned in this case.

Evaluating a model

The `random_split` function provides a safe way to split the `observation_data` into a train and validation split specially for the task of churn prediction. The recommended way to evaluate a churn prediction model is to first simulate what the model would have predicted at a `time_boundary` in the past and compare those predictions with the ground truth obtained from events after the `time_boundary`. This can be done as follows:

```
eval_time = datetime.datetime(2011, 10, 1)
metrics = model.evaluate(valid, time_boundary = eval_time)
```

```
{
    'auc'      : 0.6634142545907242,
    'recall'   : 0.6243386243386243,
    'precision': 0.6310160427807486,
    'evaluation_data':
        +-----+-----+-----+
        | CustomerID | probability | label |
        +-----+-----+-----+
        | 12348     | 0.93458378315 | 1   |
        | 12361     | 0.437742382288 | 1   |
        | 12365     | 0.5           | 1   |
        | 12375     | 0.769197463989 | 0   |
        | 12380     | 0.339888572693 | 0   |
        | 12418     | 0.15767210722 | 1   |
        | 12432     | 0.419652849436 | 0   |
        | 12434     | 0.88883471489 | 1   |
        | 12520     | 0.0719764530659| 1   |
        | 12546     | 0.949095606804 | 0   |
        +-----+-----+-----+
        [359 rows x 3 columns]
    'roc_curve':
        +-----+-----+-----+-----+
        | threshold | fpr | tpr | p  | n  |
        +-----+-----+-----+-----+
        | 0.0       | 1.0 | 1.0 | 189 | 170 |
        | 1e-05     | 1.0 | 1.0 | 189 | 170 |
        | 2e-05     | 1.0 | 1.0 | 189 | 170 |
        | 3e-05     | 1.0 | 1.0 | 189 | 170 |
        | 4e-05     | 1.0 | 1.0 | 189 | 170 |
        | 5e-05     | 1.0 | 1.0 | 189 | 170 |
        | 6e-05     | 1.0 | 1.0 | 189 | 170 |
        | 7e-05     | 1.0 | 1.0 | 189 | 170 |
        | 8e-05     | 1.0 | 1.0 | 189 | 170 |
        | 9e-05     | 1.0 | 1.0 | 189 | 170 |
        +-----+-----+-----+-----+
        [100001 rows x 5 columns]
    'precision_recall_curve':
        +-----+-----+-----+
        | cutoffs | precision | recall  |
        +-----+-----+-----+
        | 0.1     | 0.568181818182 | 0.925925925926 |
        | 0.25    | 0.6138996139   | 0.84126984127  |
        | 0.5     | 0.631016042781 | 0.624338624339 |
        | 0.75    | 0.741935483871 | 0.243386243386 |
        | 0.9     | 0.533333333333 | 0.042328042328 |
        +-----+-----+-----+
        [5 rows x 3 columns]
}
```

Accessing the underlying features & model

After a churn prediction model is trained, one can access the underlying features and boosted tree model that are being used:

```
# Get the trained boosted trees model.  
bt_model = model.trained_model  
  
# Get the training data after feature engineering.  
train_data = model.processed_training_data
```

Explaining predictions

In addition to model predictions, one can also obtain explanations about why the model made a specific prediction. This can be obtained with the `explain` method whose API is very similar to the `predict` method.

```
eval_time = datetime.datetime(2011, 10, 1)  
explanation = model.explain(valid, time_boundary = eval_time)  
print explanation['explanation'][0]
```

```
['Less than 6.50 days since the first event in the last 90 days',  
'Sum of "Quantity" in the last 90 days more than (or equal to) 200.50',  
'Less than 2.50 days between the first two events in the last 90 days',  
'Sum of "Quantity" in the last 21 days less than 547.00',  
'More than (or equal to) 1.50 "min" events in feature "Quantity" each day in the last 90',  
'Less than 111.00 "sum" events in feature "Quantity" in the last 60 day']
```

These are explanations that are specific to each user. The `get_feature_importance` function can provide an overview of the set of features that are important to this model while making predictions.

```
print model.get_feature_importance()
```

name	index	count	description
Quantity features 7	user_timesinceseen	99	Days since most recent event
Quantity features 90	sum sum	43	Sum of "Quantity" in the 1...
Quantity features 90	sum max	20	Max of "Quantity" in the 1...
Quantity features 60	sum min	20	Sum of "Quantity" is each...
Quantity features 60	sum sum	20	Sum of "Quantity" in the 1...
Quantity features 60	count sum	17	"sum" events in feature "Q...
Quantity features 90	sum count	17	Days with an event in the ...
Quantity features 60	count ratio	16	Average number of "ratio" ...
Quantity features 90	count sum	16	"sum" events in feature "Q...
Quantity features 90	sum min	16	Sum of "Quantity" is each...

Making a churn report

In addition to predictions and explanations, one can also get a detailed churn report which clusters the users into segments and then provides a reason for churn for users in the segment.

```
report = model.get_churn_report(valid)
```

The output of the churn report is an SFrame which segments users that have a similar churn probability and reason for churn.

Visualizations

Once a model has been trained, you can easily visualize the model. There are three built-in visualizations to help explore, explain, and evaluate the model:

- **Explore:** This view helps explore the predictions (and associated explanations) made by the model as generated by the churn report.
- **Evaluate:** This view helps quickly provide all the information needed to determine if the model compares well with a simple activity based baseline.
- **Overview:** This view combines the explore and evaluate views into a tabbed view.

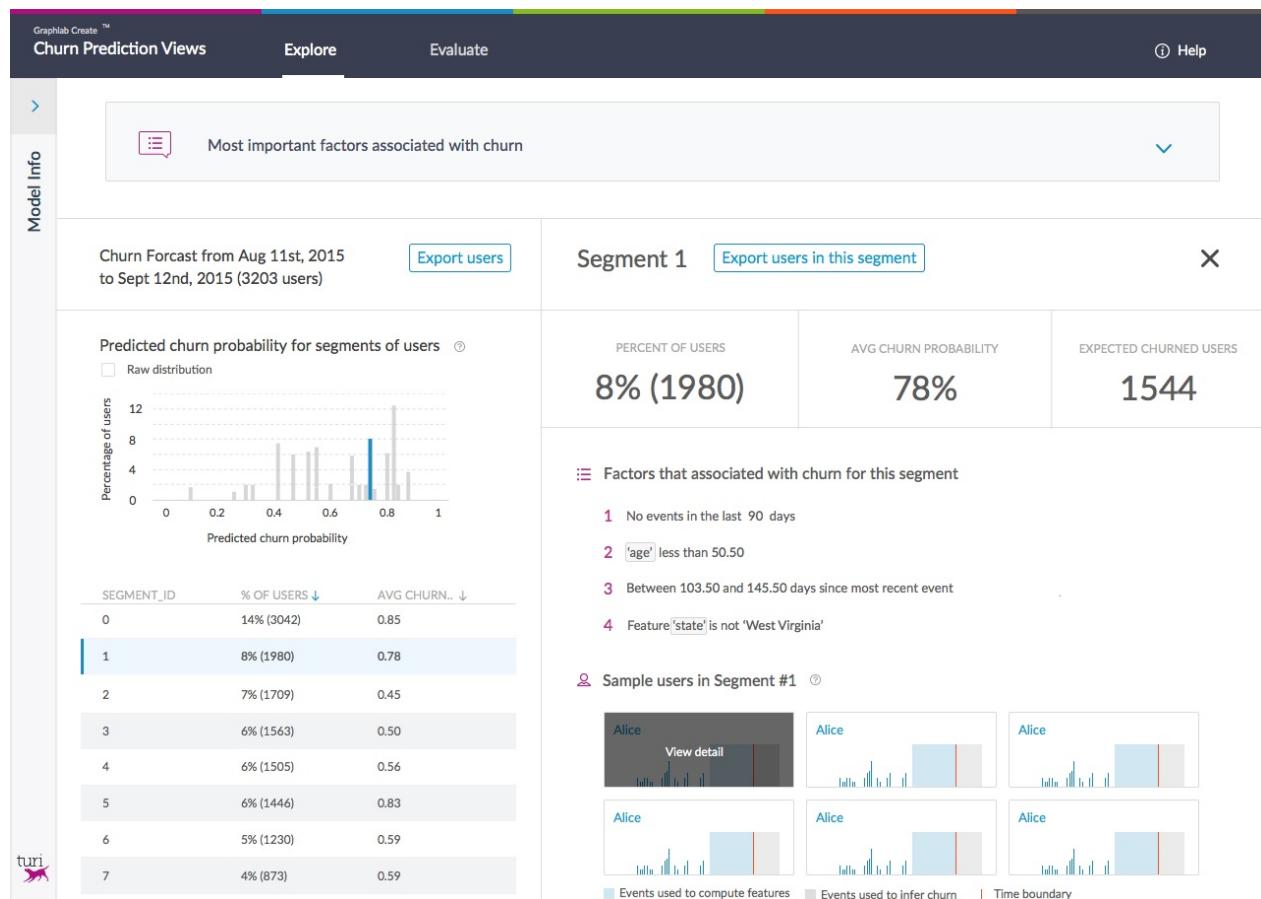
Explore view

Once a churn prediction model has been trained, an interactive visualization component can help **explore** the predictions made by the model. The explore view contains a collection of useful interactive visuals that can help explore:

- The distribution of predicted churn probability for users.
- A useful segmentation of users that have similar characteristics and churn probability.
- Event time lines, user features, predictions and corresponding explanations for a representative random sample of users.

The explore view can be obtained as follows:

```
time_boundary = datetime.datetime(2011, 10, 1)
view = model.views.explore(train, time_boundary)
view.show()
```



Evaluate View

The evaluate view creates a visualization that compares the model to a baseline. The default baseline (called the activity model) is a model trained using logistic regression using **only** the time duration of in-activity to predict churn.

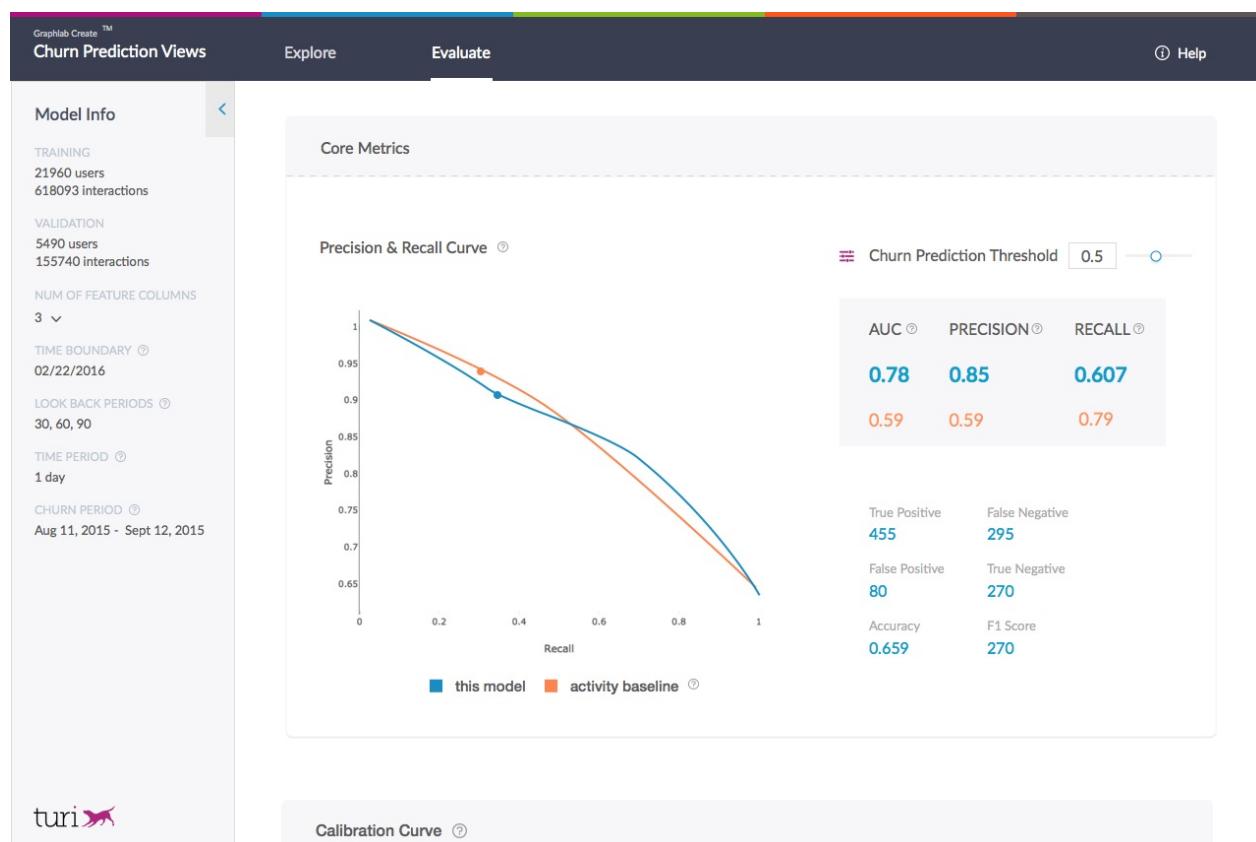
The evaluate view contains the following collection of useful interactive visuals:

- A precision-recall curve to illustrate the trade-off between the ability to correctly predict churned users (precision) & the ability to find all users who might churn (recall). The curve also compares the trained model with the activity baseline.

- A calibration curve that compares the true probability of an event with its predicted probability. This chart shows if the trained model has well "calibrated" probability predictions. The closer the calibration curve is to the "ideal" curve, the more confident one can be in interpreting the probability predictions as a confidence score for predicting churn.

The evaluate view can be obtained as follows:

```
time_boundary = datetime.datetime(2011, 10, 1)
view = model.views.evaluate(train, time_boundary)
view.show()
```



Overview

The **overview** is a tabbed view that combines the explore and evaluate.

```
time_boundary = datetime.datetime(2011, 10, 1)
view = model.views.overview(train, time_boundary)
view.show()
```

Saving and loading a model

The model can be saved for later use, either on the local machine or in an AWS S3 bucket. The saved model sits in its own directory, and can be loaded back in later to make more predictions.

```
# Save the model for later use
model.save("my_model")

# Load the model!
model = graphlab.load_model("my_model")
```

Alternate data formats

The churn predictor model supports data with 3 formats

- Raw event logs
- Aggregated logs
- User side data

Aggregated event logs

The churn prediction model performs a series of feature engineering steps. The first of the many feature transformations involves a re-sample operation which aggregates data into a fixed granularity/time-scale e.g. daily, weekly, or monthly (defined by `time_period`). If your data is already aggregated at a granularity level that is of interest to you, then you can skip this step with the option `is_data_aggregated = True`.

CustomerID	InvoiceDate	Quantity	UnitPrice
17850	2010-12-01 00:00:00	8.62741312741	4.15194658945
17850	2010-12-02 00:00:00	9.96823138928	3.23171171171
17850	2010-12-03 00:00:00	6.73478655767	5.04727066303
17850	2010-12-04 00:00:00	None	None
17850	2010-12-05 00:00:00	6.01651376147	2.89657614679
17850	2010-12-06 00:00:00	5.52320783909	4.55873646209
17850	2010-12-07 00:00:00	8.43570705366	28.7384711441
17850	2010-12-08 00:00:00	8.59123536079	3.76957310162
17850	2010-12-09 00:00:00	6.37530266344	5.07181943964
13047	2010-12-10 00:00:00	7.35931834663	4.65013052937

The above data is already aggregated by day. This data can be used as is while training the model.

```
# Train a churn prediction model.
model = gl.churn_predictor.create(train, user_id='CustomerID',
                                    features = ['Quantity'],
                                    churn_period = churn_period,
                                    is_data_aggregated = True)
```

Using user side data

In many cases, additional metadata about the users can improve the quality of the predictions. This includes geographic location, age, profession etc. We call this type of information user side data.

The `user_data` parameter is an SFrame and must have a user column that corresponds to the `user_id` column in the `observation_data`. The churn prediction toolkit will automatically incorporate the user side data while training the model.

```
side_data = gl.SFrame(  
    'https://static.turi.com/datasets/churn-prediction/online_retail_side_data.csv')  
  
# Train a churn prediction model.  
model = gl.churn_predictor.create(train, user_id='CustomerID',  
    features = ['Quantity'],  
    churn_period = churn_period,  
    user_data = side_data)  
  
# Make predictions  
predictions = model.predict(valid, user_data = side_data)  
  
# Evaluate the model  
evaluation_time = datetime.datetime(2011, 10, 1)  
metrics = model.evaluate(valid, evaluation_time, user_data = side_data)
```

Note: Once a model is trained with user side information, the model does not store a copy of the user side data. That must be provided during every `predict` and `evaluate` call.

How does this model work?

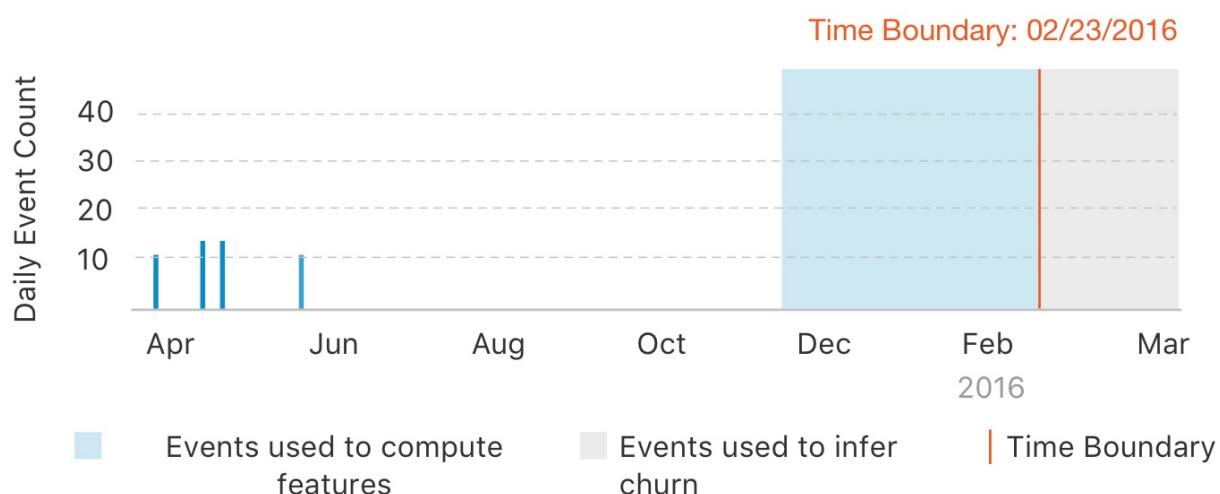
There two stages during the model training phase:

- **Phase 1:** Feature engineering. In this phase, features are generated using the provided activity data. For this phase, only the data before the provided `time_boundary` is used. The data after the `time_boundary` is used to infer the prediction target (labels).
- **Phase 2:** Machine learning model training. In this phase, the computed features and the inferred labels are used to train a classifier model (using boosted trees).

Phase 1: Feature engineering

For **Phase 1**, this toolkit performs a series of extremely rich set of feature transformations based on:

- aggregate statistics (over various periods of time) of the raw input feature columns.
- patterns over various period of time (e.g. rate of change of aggregate usage).
- user metadata (using the `user_data` parameter),



The following table provides a description of all the engineered features used in the model (T is a parameter that can be controlled using the parameter `lookback_periods`). Hence, these features are computed for each value of T in `lookback_periods`.

Engineered features
Average number of events in the last T days
Average value of a feature in the last T days
Largest daily value of a feature in the last T days
Largest time interval between two events in the last T days
Number of days since the first event in the last T days
Number of days since the most recent event in the last T days
Number of days to the most recent event in the last T days
Number of days with an event in the last T days
Number of days with an event in the last T days
Number of events in the last T days
Smallest daily value in the last T days
Smallest time interval between two events in the last T days
Sum of a column in the last T days
T day trend in the number of events
Time interval between the first two events in the last T days
Time interval between the two most recent events in the last T days

Phase 2: Model trianing

For **Phase 2**, a classifier model is trained using gradient boosted trees. Note that a churn prediction model can be trained without any labelled data. All the target labels required for training the boosted tree model are inferred based on the activity data from the past. For example, a dataset that contains data from January 2011 to December 2011 contains historical information about whether or or a user churned during each of the months prior to November 2011.

For a given `time_boundary` (say October 2011), all the events in `observation_data` `after` October 2011 are not (and must never be) included in the training data for the model. In order to create more training data for the boosted tree classifier, multiple time-boundaries can be used (using the parameter `time_boundaries`).

Frequent Pattern Mining

A frequent pattern is a substructure that appears frequently in a dataset. Finding the frequent patterns of a dataset is a essential step in data mining tasks such as feature extraction and association rule learning. The frequent pattern mining toolkit provides tools for extracting and analyzing frequent patterns in pattern data.

Introductory Example

Let us look a simple example of receipt data from a bakery. The dataset consists of items like *ApplePie* and *GanacheCookie*. The task is to identify sets of items that are frequently bought together. The dataset consists of 266209 rows and 6 columns which look like the following. The dataset was constructed by modifying the [Extended BAKERY dataset..](#)

Data:

Receipt	SaleDate	EmpId	StoreNum	Quantity	Item
1	12-JAN-2000	20	20	1	GanacheCookie
1	12-JAN-2000	20	20	5	ApplePie
2	15-JAN-2000	35	10	1	CoffeeEclair
2	15-JAN-2000	35	10	3	ApplePie
2	15-JAN-2000	35	10	4	AlmondTwist
2	15-JAN-2000	35	10	3	HotCoffee
3	8-JAN-2000	13	13	5	OperaCake
3	8-JAN-2000	13	13	3	OrangeJuice
3	8-JAN-2000	13	13	3	CheeseCroissant
4	24-JAN-2000	16	16	1	TruffleCake

[266209 rows x 6 columns]

In order to run a frequent pattern mining algorithm, we require an **item columns**, (the column *Item* in this example), and a set of **feature columns** that uniquely identify a transaction (the column *Receipt* in this example).

In just a few lines of code we can do the following:

- Find the most frequently occurring patters satisfying various conditions.
- Extract features from the dataset by transforming it from the *Item* space into the *Receipt* space. These features can then be used for applications like clustering, classification, churn prediction, recommender systems etc.
- Make predictions based on new data using rules learned from sets of items that occur frequently together.

Here is a simple end-to-end example:

```
import graphlab as gl

# Load the dataset
train = gl.SFrame("https://static.turi.com/datasets/extended-bakery/bakery_train.sf")
test = gl.SFrame("https://static.turi.com/datasets/extended-bakery/bakery_test.sf")

# Make a train-test split.
train, test = bakery_sf.random_split(0.8)

# Build a frequent pattern miner model.
model = gl.frequent_pattern_mining.create(train, 'Item',
                                             features=['Receipt'], min_length=4, max_patterns=500)

# Obtain the most frequent patterns.
patterns = model.get_frequent_patterns()

# Extract features from the dataset and use in other models!
features = model.extract_features(train)

# Make predictions based on frequent patterns.
predictions = model.predict(test)
```

Interpreting Results

Frequent pattern mining can provide valuable insight about the sets of items that occur frequently together. When a model is trained, the `model.summary()` output shows the most frequently occurring patterns together.

```
patterns = model.get_frequent_patterns()
print patterns
```

	pattern	support
	[CoffeeEclair, HotCoffee, ApplePie, AlmondTwist]	1671
	[LemonCookie, LemonLemonade, RaspberryCookie, RaspberryL...]	1550
	[LemonLemonade, RaspberryCookie, RaspberryLemonade, Gree...]	1257
	[LemonCookie, LemonLemonade, RaspberryCookie, RaspberryL...]	1256
	[AppleTart, AppleDanish, AppleCroissant, CherrySoda]	1227
	[CherryTart, ApricotDanish, OperaCake, ApricotTart]	58
	[CherryTart, ApricotDanish, OperaCake, AppleDanish]	56
	[CherryTart, ApricotDanish, GongolaisCookie, OperaCake]	54
	[CherryTart, ApricotDanish, OperaCake, VanillaEclair]	53
	[CherryTart, ApricotDanish, OperaCake, LemonLemonade]	53

[500 rows x 2 columns]

Note that the **pattern** column contains the patterns that occur frequently together and the **support** column contains the number of times these patterns occur together in the entire dataset. In this example, the pattern *[CoffeeEclair, HotCoffee, ApplePie, AlmondTwist]* occurred 860 times in the training data.

A **frequent pattern** is a set of items with a support greater than user-specified **minimum support** threshold. However, there is significant redundancy in mining frequent patterns; every subset of a frequent pattern is also frequent (e.g. *CoffeeEclair* must be frequent if *CoffeeEclair, HotCoffee* is frequent). The frequent pattern mining toolkit avoids this redundancy by mining the **closed frequent patterns**: frequent patterns with no superset of the same support.

Minimum support

One can change the minimum support above which patterns are considered frequent using the **min_support** setting:

```
model = gl.frequent_pattern_mining.create(train, 'Item',
                                           features=['Receipt'], min_support = 5000)
print model
```

```

Class : FrequentPatternMiner

Model fields
-----
Min support : 5000
Max patterns : 100
Min pattern length : 1

Most frequent patterns
-----
['CoffeeEclair'] : 6582
['HotCoffee'] : 6131
['TuileCookie'] : 6011
['StrawberryCake'] : 5624
['CherryTart'] : 5613
['ApricotDanish'] : 5582
['OrangeJuice'] : 5495
['GongolaisCookie'] : 5437
['MarzipanCookie'] : 5378
['BerryTart'] : 5087

```

Top-k frequent patterns.

In practice, we rarely know the appropriate *min_support* threshold to use. As an alternative to specifying a minimum support, we can specify a maximum number of patterns to mine using the **max_patterns** parameter. Instead of mining all patterns above a minimum support threshold, we mine the most frequent patterns until the maximum number of closed patterns are found. For large data sets, this mining process can be time-consuming. We recommend specifying a large initial minimum support bound to speed up the mining.

```

model = gl.frequent_pattern_mining.create(train, 'Item',
                                           features=['Receipt'], max_patterns = 5)
print model

```

```

Class : FrequentPatternMiner

Model fields
-----
Min support      : 1
Max patterns     : 5
Min pattern length : 1

Most frequent patterns
-----
['CoffeeEclair']      : 6582
['HotCoffee']          : 6131
['TuileCookie']        : 6011
['StrawberryCake']     : 5624
['CherryTart']         : 5613

```

Note: The algorithm for extracting the top-k most frequent occurring patterns can be severely sped up with a good estimate for the lower bound on *min_support*.

Minimum Length

Typically, the most frequent patterns are of length 1. However, in practice, patterns of length 1 may not very useful. To mine patterns greater than a minimum length, we use the **min_length** parameter:

```

model = gl.frequent_pattern_mining.create(train, 'Item',
                                           features=['Receipt'], min_length = 5)
print model

```

```

Class : FrequentPatternMiner

Model fields
-----
Min support : 1
Max patterns : 100
Min pattern length : 5

Most frequent patterns
-----
['LemonCookie', 'LemonLemonade', 'RaspberryCookie', 'RaspberryLemonade', 'GreenTea']: 125
['CoffeeEclair', 'HotCoffee', 'VanillaFrappuccino', 'ApplePie', 'AlmondTwist']: 21
['CoffeeEclair', 'HotCoffee', 'ApplePie', 'AlmondTwist', 'VanillaMeringue']: 21
['CoffeeEclair', 'HotCoffee', 'ApplePie', 'AlmondTwist', 'LemonTart']: 20
['CoffeeEclair', 'HotCoffee', 'NapoleonCake', 'ApplePie', 'AlmondTwist']: 19
['CoffeeEclair', 'HotCoffee', 'MarzipanCookie', 'ApplePie', 'AlmondTwist']: 17
['CoffeeEclair', 'HotCoffee', 'ApplePie', 'AlmondTwist', 'CherrySoda']: 17
['CoffeeEclair', 'HotCoffee', 'ApplePie', 'AlmondTwist', 'LemonLemonade']: 17
['CoffeeEclair', 'HotCoffee', 'GongolaisCookie', 'ApplePie', 'AlmondTwist']: 16
['CoffeeEclair', 'HotCoffee', 'CherryTart', 'ApplePie', 'AlmondTwist']: 16

```

Note: The three parameters *min_support*, *max_patterns*, and *min_length* can be combined to find patterns satisfying all conditions.

Extracting Features

Using the set of closed patterns, we can convert pattern data to binary features vectors. These feature vectors can be used for other machine learning tasks, such as clustering or classification. For each input pattern x , the j-th extracted feature $f_x[j]$ is a binary indicator of whether the j-th closed pattern is contained in x .

```

model = gl.frequent_pattern_mining.create(train, 'Item',
                                           features=['Receipt'])
features = model.extract_features(test)

```

```

Columns:
    Receipt      int
    extracted_features      array

Rows: 15000

Data:
+-----+-----+
| Receipt |      extracted_features |
+-----+-----+
| 63664 | [0.0, 0.0, 0.0, 0.0, 0.0, ... |
| 62361 | [0.0, 0.0, 0.0, 0.0, 0.0, ... |
| 66110 | [0.0, 0.0, 1.0, 0.0, 0.0, ... |
| 61406 | [0.0, 0.0, 0.0, 0.0, 0.0, ... |
| 69188 | [0.0, 0.0, 0.0, 0.0, 0.0, ... |
| 65762 | [0.0, 0.0, 0.0, 0.0, 0.0, ... |
| 74562 | [0.0, 1.0, 0.0, 0.0, 0.0, ... |
| 66750 | [0.0, 0.0, 0.0, 0.0, 0.0, ... |
| 60908 | [1.0, 0.0, 0.0, 0.0, 0.0, ... |
| 62213 | [0.0, 1.0, 0.0, 1.0, 0.0, ... |
+-----+
[15000 rows x 2 columns]

```

Once the features are extracted, we can use them downstream in other applications such as clustering, classification, churn prediction, recommender systems etc.

Making Predictions

An **association rule** is an ordered pair of item sets (prefix A , prediction B) denoted $A \Rightarrow B$ such that A, B are disjoint and $A \cup B$ is frequent. Because every frequent pattern generates multiple association rules (a rule for each subset), we evaluate and filter rules using a score criteria. The most popular criteria for scoring association rules is to measure the **confidence** of the rule: the ratio of the support of $A \cup B$ to the support of A . The **confidence** of the rule $A \Rightarrow B$ is our empirical estimate of the conditional probability for B given A :

$$\text{Confidence}(A \Rightarrow B) = \frac{\text{Supp}(A \cup B)}{\text{Supp}(A)}$$

One can make predictions using the *predict* or *predict_topk* method for single and multiple predictions respectively. The output of both the methods is an SFrame with the following columns:

- **prefix**: The *antecedent* or *left-hand side* of an association rule. It must be a frequent pattern and a subset of the associated pattern.
- **prediction**: The *consequent* or *right-hand side* of the association rule. It must be disjoint of the prefix.

- **confidence:** The confidence of the association rule as defined above.
- **prefix support:** The frequency of the *prefix* pattern in the training data.
- **joint support:** The frequency of the co-occurrence (*prefix + prediction*) in the training data

If no valid association rule exists for an pattern, then `predict` will return a row of Nones.

```
predictions = model.predict(test)
```

Columns:

Receipt	int
prefix	list
prediction	list
confidence	float
prefix support	int
joint support	int

Rows: 15000

Data:

Receipt	prefix	prediction	confidence	prefix support
63664	[]	[CoffeeEclair]	0.109701828364	59999
62361	[OperaCake]	[CherryTart]	0.531376518219	4940
66110	[ApricotDanish]	[CherryTart]	0.574883554282	5582
61406	[]	[CoffeeEclair]	0.109701828364	59999
69188	[ApricotDanish]	[CherryTart]	0.574883554282	5582
65762	[]	[CoffeeEclair]	0.109701828364	59999
74562	[HotCoffee]	[CoffeeEclair]	0.3069646061	6131
66750	[OperaCake]	[CherryTart]	0.531376518219	4940
60908	[MarzipanCookie]	[TuileCookie]	0.5621048717	5378
62213	[StrawberryCake]	[NapoleonCake]	0.464971550498	5624
<hr/>				
<hr/>				
joint support				
<hr/>				
6582				
2625				
3209				
6582				
3209				
6582				
1882				
2625				
3023				
2615				
<hr/>				
[15001 rows x 6 columns]				

Note: If the number of patterns extracted is large, then prediction could potentially be a slow operation.

Accessing Model Attributes

We will now go over some more advanced options with the frequent pattern mining module. This includes advanced options for pattern mining, model interpretation, extracting features, and making predictions via rule mining.

The attributes of all GraphLab Create models, which include training statistics, model hyper-parameters, and model results can be accessed in the same way as python dictionaries. To get a list of all fields that can be accessed, you can use the `list_fields()` function:

```
fields = model.list_fields()
print fields
['features',
 'frequent_patterns',
 'item',
 'max_patterns',
 'min_length',
 'min_support',
 'num_examples',
 'num_features',
 'num_frequent_patterns',
 'num_items',
 'training_time']
```

Each of these fields can be accessed using dictionary-like lookups. For example, the `num_frequent_patterns` is the number of frequent patterns extracted by the model.

```
model['num_frequent_patterns']
500
```

The [API docs](#) provide a detailed description of each of the model attributes.

References

- Han, Jiawei, et al. *Frequent pattern mining: current status and future directions*. Data Mining and Knowledge Discovery 15.1 (2007): 55-86.
- Han, Jiawei, Micheline Kamber, and Jian Pei. *Data mining: concepts and techniques: concepts and techniques*. Elsevier, 2011.
- Wang, Jianyong, et al. *TFP: An efficient algorithm for mining top-k frequent closed patterns*. Knowledge and Data Engineering, IEEE Transactions on 17.5 (2005): 652-

663.

Sentiment analysis

Data scientists are often faced with data sets that contain text, and must employ natural language processing (NLP) techniques in order to make it useful. [Sentiment analysis](#) refers to the use of NLP techniques to extract subjective information such as the polarity of the text, e.g., whether or not the author is speaking positively or negatively about some topic.

In many cases, sentiment analysis can help keep a pulse on the users' needs and adapt the product and services accordingly. Many applications exist for this type of analysis:

- **Forum data:** Find out how people feel about various products and features.
- **Restaurant and movie reviews:** What are people raving about? What do people hate?
- **Social media:** What is the sentiment about a hashtag, e.g. for a company, politician, etc?
- **Call center transcripts:** Are callers praising or complaining about particular topics?

In the next section, you will learn to use GraphLab Create's [sentiment_analysis](#) toolkit to apply pre-trained models to predict sentiment for text data in these situations.

More advanced forms of sentiment analysis exist. Aspect mining attempts to identify features (or aspects) of entities that are mentioned, and then estimate the sentiment for each aspect. For example, when studying reviews about mobile phones you may be interested in how people feel about aspects such as battery life, screen resolution, size, etc.

For these situations we provide a [product_sentiment](#) toolkit where it's easy to explore and summarize sentiment about products within text data. The toolkit enables to search for aspects of interest and obtain summaries of the reviews or sentences with the most positive (or negative) predicted sentiment.

Sentiment Analysis

This toolkit allows you to take text and predict whether it contains positive or negative sentiment. For instance, the model will predict "positive sentiment" for a snippet of text -- whether it is a movie review or a tweet -- when it contains words like "awesome" and "fantastic". Likewise, having many words with a negative connotation will yield a prediction of "negative sentiment".

```
>>> import graphlab as gl
>>> data = gl.SFrame({'text': ['hate it', 'love it']})
>>> m = gl.sentiment_analysis.create(data, features=['text'])
>>> m.predict(data)
dtype: float
Rows: 2
[0.5565335646003744, 0.7532539958283951]
```

Notice that scores closer to 1 represent "positive sentiment", while scores near 0 represent "negative sentiment". These predictions are made using a statistical model trained on review data, and can be a useful starting point for your application.

Working with SentimentAnalysisModels

A SentimentAnalysisModel is currently a simple combination of two components:

- feature engineering: a [bag-of-words](#) transformation
- statistical model: a LogisticClassifier is used to score whether the text contains positive or negative sentiment

After creating the above model, you can inspect it via

```
>>> m
Class : SentimentAnalysisModel

Data
-----
Number of rows : 2

Model
-----
Score column : None
Features : ['text']
Method : bow-logistic
```

You can list the available attributes via `m.list_fields()`. You will see that the two important internal components can be inspected and used.

```
# Obtain the function used to process text for the internal classifier.
>>> f = m['feature_extractor']
>>> f(data)
Out[15]:
Columns:
    bow      dict

Rows: 2

Data:
+-----+
|      bow      |
+-----+
| {'hate': 1, 'it': 1} |
| {'love': 1, 'it': 1} |
+-----+
[2 rows x 1 columns]

# Obtain the internal classifier object.
>>> print m['classifier']
Class                      : LogisticClassifier

Schema
-----
Number of coefficients     : 2278286
Number of examples          : 1267133
Number of classes            : 2
Number of feature columns    : 1
Number of unpacked features   : 2278285

Hyperparameters
-----
L1 penalty                  : 0.0
L2 penalty                  : 0.05

Training Summary
-----
Solver                      : auto
Solver iterations             : 10
Solver status                 : TERMINATED: Iteration limit reached.
Training time (sec)           : 39.9574

Settings
-----
Log-likelihood                : 68620.2104

Highest Positive Coefficients
-----
bow[streaming/gaming.]       : 15.1311
```

```

bow[whole....]           : 15.0669
bow[2bd/3bath,]          : 13.9028
bow[later...there]        : 13.6987
bow[cris']               : 13.6801

Lowest Negative Coefficients
-----
bow[$136.]              : -16.0415
bow[directions....just]   : -15.4148
bow[fathom!]              : -15.1889
bow["chick-fil-a"]        : -14.8406
bow[bece]                 : -14.2866

```

Details about Turi's pre-trained models

When a target column name is not provided, a pre-trained model will be used.

Predicted scores from the current model are between 0 and 1, where higher scores indicate more positive predicted sentiment. The model is a

`graphlab.logistic_classifier.LogisticClassifier` model that can be obtained via
`m['classifier']`.

The first time this feature is used, GLC will download the model from a public S3 bucket to a local temporary directory within `~/.graphlab`. Future calls to the toolkit will attempt to use the cached version to avoid having to redownload the model.

Data sets: The model is currently trained on review data that includes:

- a random 20% subset of the [Amazon product data](#) (collected by Julian McAuley) with "review/score" as the target and "review/text" as the review.
- Yelp data (version 5), using "stars" as the target.

Preprocessing: Feature engineering includes a bag-of-words representation of the text data. Both data sets have an integer "score" column with values ranging between 1 and 5, representing the number of stars that the user gave. Scores less than 3 are given a target of 0 (and thus considered to be negative sentiment during training) and ratings of more than 3 are considered positive sentiment. Reviews with a score of 3 are currently considered neutral and removed prior to training the classifier.

Accuracy: Given the above preprocessing, we observe validation set accuracies between 88-90% on both of these data sets. More details forthcoming.

Known issues: The model is currently tuned to focus on English-only text corpuses, and works best when each piece of text has similar length.

Versioning: These models may change across versions of GLC.

Training your own sentiment classifier

You may also train your own model. This can be useful when you want to tune the model to your data set to take advantage of vocabulary that is particular to your application, or you are attempting to predict something such as "utility" by training on whether users thought a review was useful.

```
>>> import graphlab as gl
>>> data = gl.SFrame({'rating': [1, 5], 'text': ['hate it', 'love it']})
>>> m = gl.sentiment_analysis.create(data, 'rating', features=['text'])
>>> m.predict_row({'text': 'really love it'})
>>> m.predict_row({'text': 'really hate it'})
```

FAQ

Why use bag-of-words and a logistic regression classifier? Because this combination is a very strong baseline for this particular task. Getting other approaches to match its speed/accuracy is difficult, but we will be pursuing this in the future.

Product sentiment

This toolkit aims to help you explore and summarize sentiment about products within text data. The toolkit enables to search for aspects of interest and obtain summaries of the reviews or sentences with the most positive (or negative) predicted sentiment.

Summarizing sentiment

As a quick example, suppose we want to summarize people's Comcast complaints. We can use `graphlab.product_sentiment.create` to build a model that can be queried for summaries of product sentiment:

```
>>> import graphlab as gl
>>> sf = gl.SFrame('https://static.turi.com/datasets/comcast_fcc_complaints_apr_june_2015')
>>> m = gl.product_sentiment.create(sf, features=['Description'], splitby='sentence')
>>> m.sentiment_summary(['billing', 'cable', 'cost', 'late', 'charges', 'slow'])
+-----+-----+-----+
| keyword | sd_sentiment | mean_sentiment | review_count |
+-----+-----+-----+
| cable   | 0.302471264675 | 0.285512408978 | 1618      |
| slow    | 0.282117103769 | 0.243490314737 | 369       |
| cost    | 0.283310577512 | 0.197087019219 | 291       |
| charges | 0.164350792173 | 0.0853637431588 | 1412      |
| late    | 0.119163914305 | 0.0712757752753 | 2202      |
| billing | 0.159655783707 | 0.0697454360245 | 583       |
+-----+-----+-----+
```

Here we see that there are 369 sentences that mention "slow", and the mean predicted sentiment is 0.243 for these. Overall, it appears the mentions of "billing" and "late" have even lower predicted sentiment.

Under the hood

In the example above, we created a `ProductSentimentModel` using the text in the "Description" column. While creating the model, several operations are completed under the hood:

- a data structure is created that helps facilitate searching for text snippets, including doing a TF-IDF transform of the text and creating an inverted index.

- each piece of text is tokenized into sentences using NLTK's punkt sentence parser.
- a pre-trained sentiment classifier scores all reviews (or sentences) and stores these scores within the model.

Providing `splitby='sentence'` argument when creating the model implies that all analysis should be performed at the sentence-level rather than using the entire text. Thus any calls to `sentiment_summary` will concern predictions for each sentence, and `get_most_positive` will return sentences rather than the entire review, for instance.

Internal components

As mentioned above, a trained model is comprised of several internal components.

To obtain the sentiment model, you may do the following. See the [sentiment_analysis](#) chapter for more details.

```
>>> m.sentiment_scorer
Class : SentimentAnalysisModel

Data
-----
Number of rows : 2224

Model
-----
Score column : None
Features : ['Description']
Method : bow-logistic
```

To obtain the model that searches for text snippets, you may do the following:

```
>>> m.review_searcher
Class : SearchModel

Corpus
-----
Number of documents : 37652
Average tokens/document : 17.8304

Indexing settings
-----
BM25 k1 : 1.5
BM25 b : 0.75
TF-IDF threshold : 0.01

Query expansion settings
-----
Number of similar tokens : 5
Maximum distance : 0.99
Near match BM25 weight : 1.0

Index
-----
Number of unique tokens indexed : 10426
Preprocessing time (s) : 8.3658
Indexing time (s) : 2.6618
```

More advanced summarization

Sometimes you may have additional information that you want to use when exploring your text data. For instance you may want the most positive/negative sentences for each unique product, each unique restaurant, etc.

The `groupby` argument allows you to provide a column name of the original data set that should be used to group the results. The most positive items will be shown for each unique value found in this column. For instance, this could be the column containing product names.

```

>>> data = gl.SFrame('https://static.turi.com/datasets/amazon_baby_products/amazon_baby.g
>>> data = data.head(10000)[['name', 'review']]
>>> m = gl.product_sentiment.create(data, features=['review'])
>>> m.get_most_negative(['cheap'], groupby='name', k=3)
+-----+-----+-----+
| __review_id | relevance_score | keyword | review |
+-----+-----+-----+
| 3315 | 8.98588339337 | cheap | I am not really impressed ... |
| 8503 | 7.742594568 | cheap | I purchased this item thin... |
| 1567 | 11.078843875 | cheap | This is a great learning t... |
| 1553 | 7.55805328553 | cheap | The dinosaurs are very goo... |
| 1929 | 5.21646511143 | cheap | With 11 years between my t... |
| 7332 | 6.16265423577 | cheap | lol. I bought this product... |
| 7235 | 10.7048436718 | cheap | I actually can get more mi... |
| 7284 | 5.10796237053 | cheap | I purchased this pump beca... |
| 2642 | 7.13302228683 | cheap | My daughter already has a ... |
| 2638 | 2.00543732422 | cheap | My sons are 6.5 months old... |
+-----+-----+-----+
+-----+-----+
| sentiment_score | name |
+-----+-----+
| 0.987693064592 | 100% Cotton Terry Contour ... |
| 0.000968652689596 | 2-in-1 Car Seat Cover `n Carry |
| 0.989658887194 | Animal Planet's Big Tub of... |
| 0.99992863105 | Animal Planet's Big Tub of... |
| 0.999956837365 | Avent ISIS Breast Pump wit... |
| 0.676748057996 | Avent Isis Manual Breast Pump |
| 0.988287661335 | Avent Isis Manual Breast Pump |
| 0.999999767378 | Avent Isis Manual Breast Pump |
| 0.899418593296 | BABYBJORN Little Potty - Red |
| 0.999999904776 | BABYBJORN Little Potty - Red |
+-----+-----+

```

Note that up to 3 reviews are shown for each unique product where the review mentioned "cheap". A `__review_id` column containing the row numbers from the original data, allowing you to find the original pieces of data corresponding to these predictions.

Anomaly Detection

Anomalies are data points that are different from other observations in some way, typically measured against a model fit to the data.

We assume the anomaly detection task is unsupervised, i.e. we don't have training data with points labeled as anomalous. Each data point passed to an anomaly detection model is given a score from 0 to infinity indicating how different the point is relative to the rest of the dataset. The calculation of this score varies between models, but a higher score always indicates a point is more anomalous. Often a threshold is chosen to make a final classification of each point as typical or anomalous; this post-processing step is left to the user.

The [GraphLab Create Anomaly Detection toolkit](#) currently includes three models for two different data contexts: **local outlier factor**, for detecting outliers in multivariate data that are assumed to be independently and identically distributed, **moving Z-score**, for scoring outliers in a univariate, sequential dataset, typically a time series, and **bayesian changepoints** for identifying changes in the mean or variance of a sequential series.

The anomaly detection toolkit is in active development, and feedback is very welcome.

Local Outlier Factor

The local outlier factor (LOF) method scores points in a multivariate dataset whose rows are assumed to be generated independently from the same probability distribution.

Background

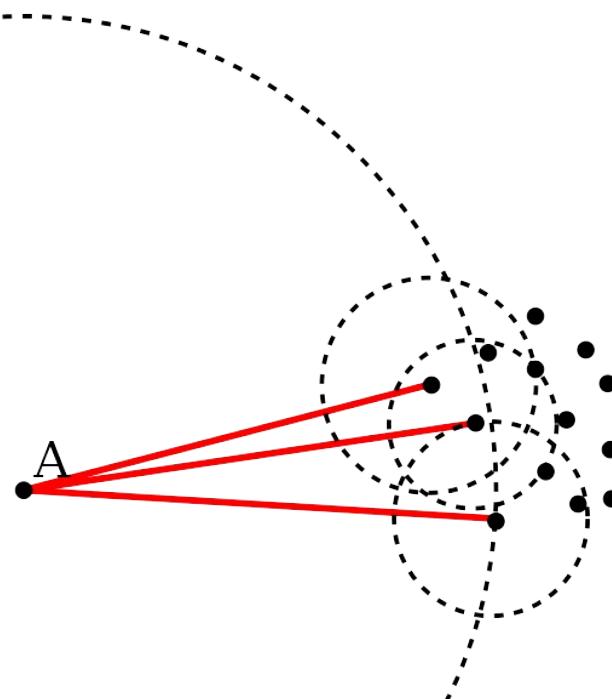
Local outlier factor is a density-based method that relies on nearest neighbors search. The LOF method scores each data point by computing the ratio of the average densities of the point's neighbors to the density of the point itself. According to the method, the estimated density of a point p is the number of p 's neighbors divided by the sum of distances to the point's neighbors.

Suppose $N(p)$ is the set of neighbors of point p , k is the number of points in this set, and $d(p, x)$ is the distance between points p and x . The estimated density is:

$$\hat{f}(p) = \frac{k}{\sum_{x \in N(p)} d(p, x)}$$

and the local outlier factor score is:

$$LOF(p) = \frac{\frac{1}{k} \sum_{x \in N(p)} \hat{f}(x)}{\hat{f}(p)}.$$



Point A has a high LOF score because its density is low relative to its neighbors' densities. Dotted circles indicate the distance to each point's third-nearest neighbor. Source: Wikipedia - Local outlier factor. File:LOF-idea.svg. <https://commons.wikimedia.org/wiki/File:LOF-idea.svg>

Creating a Local Outlier Factor Model

To show how the LOF tool is used, we'll use the customer data from the [AirBnB New User Bookings competition](#) on Kaggle. The following code assumes a copy of the file

`train_users_2.csv` is saved in the working directory. Each row in this dataset describes one of 213,451 AirBnB users; there is a mix of basic features, such as gender, age, and preferred language, and the user's "technology profile", including the browser type, device type, and sign-up method.

First, we read the data into an `SFrame`, and select a subset of features of interest. For this illustration we'll use only the basic features: gender, age, and language.

```
import graphlab as gl
users = gl.SFrame.read_csv('train_users_2.csv', header=True)

features = ['gender', 'age', 'language']
users = users[['id'] + features]
users.print_rows(5)
```

id	gender	age	language
gxn3p5htnn	-unknown-	None	en
820tgsjxq7	MALE	38.0	en
4ft3gnwmtx	FEMALE	56.0	en
bjjt8pjhuk	FEMALE	42.0	en
87mebub9p4	-unknown-	41.0	en

[213451 rows x 4 columns]

We have a small amount of data cleaning to do. I peeked ahead and realized that several users have ages of 2013 or 2014 (presumably the year was recorded accidentally), so we will re-code those values as "missing". The LOF model does not allow missing data when the model is created, so we'll actually drop any row that has a missing value (these values could also be imputed for more thorough results). After dropping low-quality rows, we're down to about 125,000 users.

```
users['age'] = users['age'].apply(lambda x: x if x < 150 else None)
users = users.dropna(columns=features, how='any')
print len(users)
```

124681

By default the LOF tool measures the difference between string-typed features with Levenshtein distance, but this is fairly slow to compute. Instead, let's use the [one-hot encoder](#) to turn the string features into categorical features. We'll also need to standardize the `age` feature so it's on roughly the same scale as the encoded categorical features.

```
encoder = gl.feature_engineering.OneHotEncoder(features=['language', 'gender'])
users2 = encoder.fit_transform(users)
users2['age'] = (users['age'] - users['age'].mean()) / users['age'].std()
```

Finally, we create the LOF model and inspect the output anomaly scores by printing the model's `scores` field.

```
model = gl.anomaly_detection.local_outlier_factor.create(users2,
                                                       features=['age', 'encoded_features'])
model['scores'].print_rows(5)
```

```
+-----+-----+-----+-----+
| row_id | density | anomaly_score | neighborhood_radius |
+-----+-----+-----+-----+
| 79732 |   inf   |      nan       |        0.0          |
| 7899  |   inf   |      nan       |        0.0          |
| 25263 |   inf   |      nan       |        0.0          |
| 87629 |   inf   |      nan       |        0.0          |
| 43116 |   inf   |      nan       |        0.0          |
+-----+-----+-----+-----+
[124681 rows x 4 columns]
```

From the `scores` SFrame we can see that the model worked successfully, scoring each of the 124,681 input rows, but we still have a bit of work to do to understand the results and make a final determination about which points are anomalies.

Note that the anomaly score for many observations in our AirBnB dataset is `nan` which indicates the point has many neighbors at exactly the same location, making the ratio of densities undefined. These points are *not* outliers.

Using the LOF Model to detect anomalies

There are two common ways to make a final decision about whether each point is typical or anomalous. The first is to simply declare the `k` points with the highest scores to be anomalous. The `topk` method of the `scores` SFrame is straightforward way to find this.

```
anomalies = model['scores'].topk('anomaly_score', k=5)
anomalies.print_rows()
```

row_id	density	anomaly_score	neighborhood_radius
119047	13.9548615034	inf	0.0716596148059
47886	13.9548615034	inf	0.0716596148059
91446	13.9548615034	inf	0.0716596148059
114118	13.9548615034	inf	0.0716596148059
21279	13.9548615034	inf	0.0716596148059

[5 rows x 4 columns]

The anomaly scores for these points are *infinite*, which happens when a point is next to several identical points, but is not itself a member of that bunch. These points are certainly anomalous, but our choice of `k` was arbitrary and excluded many points that are also likely anomalous.

A better way to get the anomalies is to find points whose anomaly score is above some *threshold*, which can be chosen by looking at the distribution of anomaly scores. The `sketch_summary` method let's us do this and find a threshold from an approximate *quantile*.

```
sketch = model['scores']['anomaly_score'].sketch_summary()
threshold = sketch.quantile(0.9)
mask = model['scores']['anomaly_score'] > threshold
anomalies = model['scores'][mask]
print "threshold:", threshold, "\nnumber of anomalies:", len(anomalies)
```

```
threshold: 8.6
number of anomalies: 173
```

To see the original features for the anomalies, we filter our input dataset to select only the anomaly rows.

```
users = users.add_row_number('row_id')
anomaly_users = users.filter_by(anomalies['row_id'], 'row_id')
anomaly_users.print_rows(20)
```

row_id	id	gender	age	language
787	w6i3ix717s	OTHER	36.0	en
4579	rpgxhr7bp2	MALE	18.0	it
4749	lnje5xn0iq	MALE	21.0	es
5328	eqsihtnz34	FEMALE	36.0	hu
6528	dyu0sssqo5	-unknown-	47.0	nl
8228	o1ciaivnyv	MALE	41.0	es
8788	91vfcvol82	MALE	91.0	en
9317	d24kga4mhu	FEMALE	39.0	de
9472	rqpt645tjk	FEMALE	39.0	de
9626	t6fvrna0t	MALE	98.0	en
10083	n45ipduv9i	MALE	28.0	fi
10727	9zhr7vpciy	MALE	39.0	fr
10765	lerui8bp4h	FEMALE	88.0	en
11038	h0cf46ubyt	MALE	27.0	fi
12293	unnvgq3efo	MALE	40.0	pl
12952	jkkz6g9y01	FEMALE	33.0	it
13926	1yoqktv6n6	OTHER	36.0	en
13980	2a9z5icq6y	MALE	39.0	de
14044	oyr9d8w1ig	OTHER	39.0	en
15761	i07vsn6wkm	FEMALE	43.0	es

[173 rows x 5 columns]

References

- Breunig, M. M., Kriegel, H., Ng, R. T., & Sander, J. (2000). [LOF: Identifying Density-Based Local Outliers](#). pp 1-12.

Moving Z-Score

The [Moving Z-score model](#) scores anomalies in a univariate sequential dataset, often a time series.

Background

The moving Z-score is a very simple model for measuring the anomalousness of each point in a sequential dataset like a time series. Given a window size w , the moving Z-score is the number of standard deviations each observation is away from the mean, where the mean and standard deviation are computed *only* over the previous w observations.

$$Z(x_i) = \frac{x_i - \bar{x}_i}{s_i}$$

where the moving mean and moving standard deviation are

For the first w observations of the series, the moving Z-score is undefined because there aren't sufficient observations to estimate the mean and standard deviation. For the GraphLab Create tool, we take the absolute value of the moving Z-score, so that the anomaly score varies from 0 to infinity, with higher scores indicating a greater degree of anomalousness.

Data and context

The Fremont bridge in Seattle is a popular place for cyclists to cross between the southern and northern halves of the city (and happens to be right next to the Turi office!). A traffic counter records the number of bicycles that cross the bridge every hour, and [the data is posted on the Seattle data portal](#). We will use the GraphLab Create Moving Z-score model to look for anomalies in this time series data.

The data can be read directly into an SFrame from the Seattle data portal's URL (the file is about 800KB) Cycle traffic is actually counted separately on each side of the bridge, but we combine these counts into a single hourly total. As a last formatting step, we convert the dataset into a [TimeSeries](#) by encoding the timestamp column as `datetime.datetime` type and setting that column as the index.

```

# Set up
import datetime as dt
import graphlab as gl

# Download data.
data_url = 'https://data.seattle.gov/api/views/65db-xm6k/rows.csv?accessType=DOWNLOAD'
hourly_counts = gl.SFrame.read_csv(data_url)

# Add the counts from each side of the bridge.
hourly_counts['count'] = (hourly_counts['Fremont Bridge West Sidewalk'] +
                           hourly_counts['Fremont Bridge East Sidewalk'])

# Convert the SFrame to a TimeSeries object
hourly_counts['timestamp'] = (hourly_counts['Date']
                               .str_to_datetime(str_format='%m/%d/%Y %H:%M:%S %P'))
hourly_counts = gl.TimeSeries(hourly_counts[['timestamp', 'count']],
                             index='timestamp')

```

By using a moving window, the The Moving Z-score model adapts well to distributional drift, but it does not handle high-frequency seasonality well. In our bicycle traffic dataset we have seasonality by hour, by day of the week, and by season of the year. In particular, the counts are low during night hours, low on the weekends, and low in the winter. We use two strategies to deal with this: first, we sum the counts with the `resample` method to get a daily traffic count; and second, we drop the weekend days entirely and focus only on weekday traffic.

```

## Group by day
daily_counts = hourly_counts.resample(dt.timedelta(1),
                                       downsample_method={'count': gl.aggregate.SUM('count')})

## Remove the weekend days
weekdays = daily_counts['timestamp'].split_datetime(column_name_prefix=None,
                                                    limit=['weekday'])
daily_counts['weekday'] = weekdays['weekday']
daily_counts = daily_counts.filter_by([5, 6], 'weekday', exclude=True)

```

Later in this chapter, we'll illustrate how Moving Z-Score models can be updated with new data. To make that more interesting, we set aside the last few months of data to use when we get to that chapter.

```

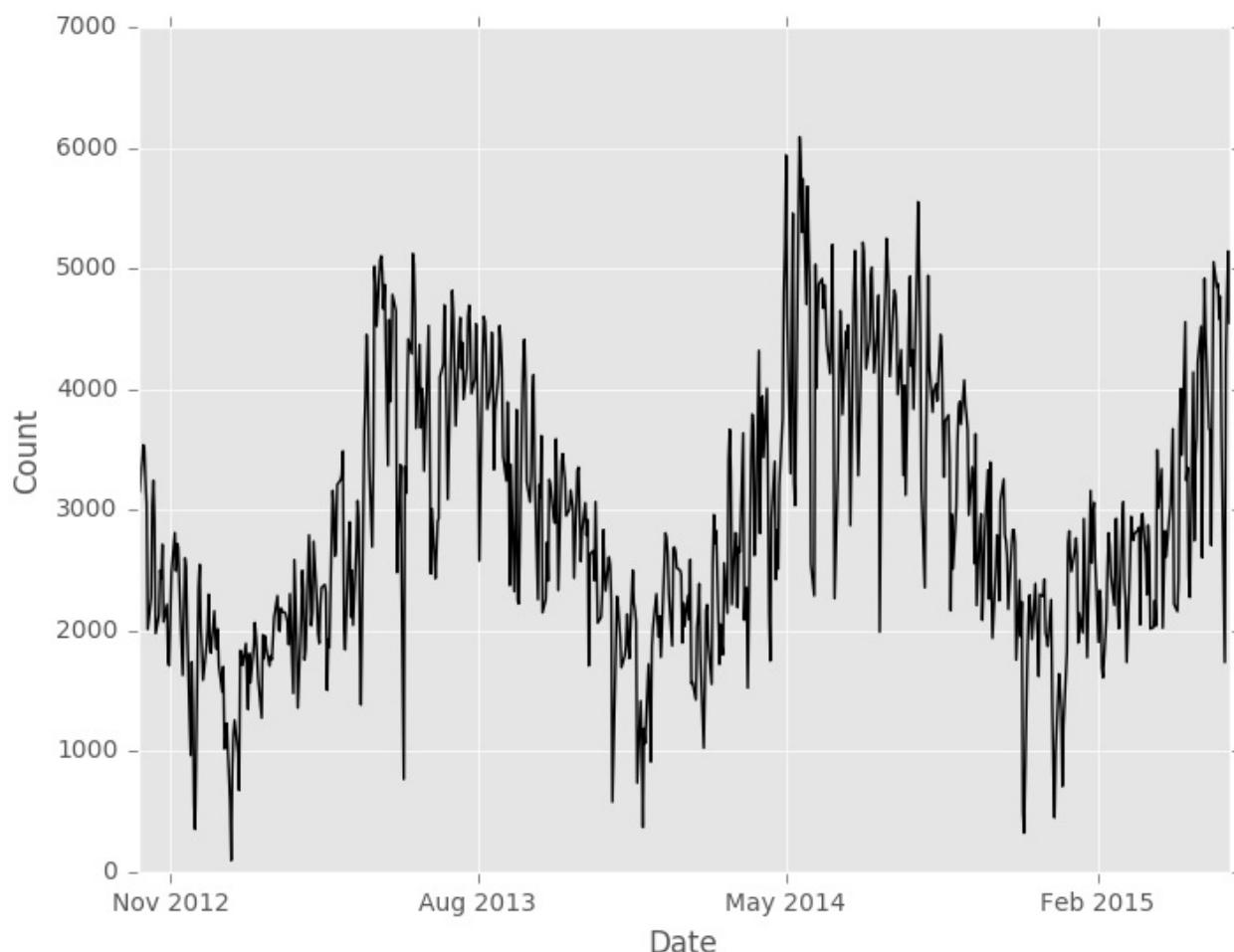
## Split dataset into original and update sets.
traffic = daily_counts[daily_counts['timestamp'] < dt.datetime(2015, 06, 01)]

```

Our final dataset `traffic` has 693 daily bicycle traffic counts, from October 3, 2012 to May 29, 2015. The data is relatively noisy to the naked eye, but clearly has some long term seasonality.

```
traffic.print_rows(5)
```

```
+-----+-----+-----+
|      timestamp      | count | weekday |
+-----+-----+-----+
| 2012-10-03 00:00:00 |  3521 |     2    |
| 2012-10-04 00:00:00 |  3475 |     3    |
| 2012-10-05 00:00:00 |  3148 |     4    |
| 2012-10-08 00:00:00 |  3537 |     0    |
| 2012-10-09 00:00:00 |  3501 |     1    |
+-----+-----+-----+
[693 rows x 3 columns]
```



Basic Moving Z-Score Usage

The Moving Z-score model takes an `SFrame` or `TimeSeries` as input (`traffic` in this case), the name of the column that contains the series to model, and the number of observations in the moving window. For this analysis our feature is the "count" column and we use a window 15 days, or three work-weeks.

```
model = gl.anomaly_detection.moving_zscore.create(traffic, feature='count',
                                                window_size=15)
```

The primary output of the Moving Z-score model is the `scores` field. This `TimeSeries` object contains the original series (`count`), original row index (`timestamp`), moving average, anomaly score, and the time the model was created (for model updating - see below). For the Moving Z-score model, the anomaly score is the absolute value of the moving Z-score. As with all GraphLab Create Anomaly Detection models, this score ranges from 0 to infinity, with higher scores indicating a greater degree of "anomalousness".

```
scores = model['scores']
scores.print_rows(20, max_row_width=100, max_column_width=20)
```

timestamp	anomaly_score	count	moving_average	model_update_time
2012-10-03 00:00:00	None	3521	None	2016-01-11 15:41...
2012-10-04 00:00:00	None	3475	None	2016-01-11 15:41...
2012-10-05 00:00:00	None	3148	None	2016-01-11 15:41...
2012-10-08 00:00:00	None	3537	None	2016-01-11 15:41...
2012-10-09 00:00:00	None	3501	None	2016-01-11 15:41...
2012-10-10 00:00:00	None	3235	None	2016-01-11 15:41...
2012-10-11 00:00:00	None	3047	None	2016-01-11 15:41...
2012-10-12 00:00:00	None	2011	None	2016-01-11 15:41...
2012-10-15 00:00:00	None	2273	None	2016-01-11 15:41...
2012-10-16 00:00:00	None	3036	None	2016-01-11 15:41...
2012-10-17 00:00:00	None	3243	None	2016-01-11 15:41...
2012-10-18 00:00:00	None	2923	None	2016-01-11 15:41...
2012-10-19 00:00:00	None	1977	None	2016-01-11 15:41...
2012-10-22 00:00:00	None	2129	None	2016-01-11 15:41...
2012-10-23 00:00:00	None	2500	None	2016-01-11 15:41...
2012-10-24 00:00:00	0.856815256628	2429	2903.73333333	2016-01-11 15:41...
2012-10-25 00:00:00	0.218502117692	2713	2830.93333333	2016-01-11 15:41...
2012-10-26 00:00:00	1.38148316083	2073	2780.13333333	2016-01-11 15:41...
2012-10-29 00:00:00	0.926826997598	2217	2708.46666667	2016-01-11 15:41...
2012-10-30 00:00:00	1.7933899856	1735	2620.46666667	2016-01-11 15:41...

[693 rows x 5 columns]

If the input dataset is an `SFrame` instead of a `TimeSeries`, the `scores` field is also an `SFrame`.

Note that the first 15 rows of the `scores` output don't have a moving average or Z-score. This is because the moving window does not have sufficient data for those observations. The `min_observations` parameter indicates the minimum number of observations needed to compute the anomaly score; by default it is the same as `window size`, but setting it to be smaller would reduce the number of missing anomaly scores, both at the beginning of the dataset and after missing values in the input data.

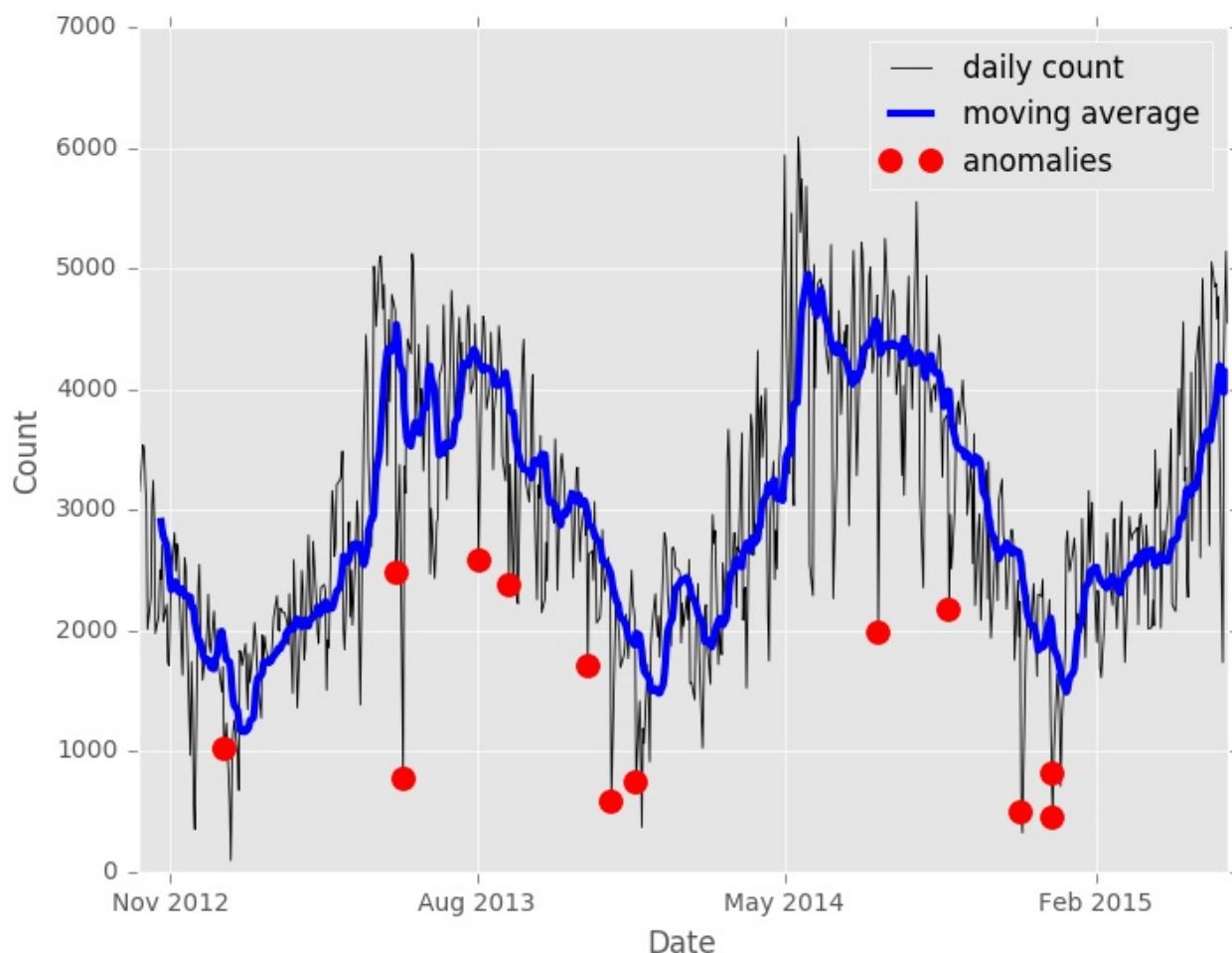
Typically the ultimate goal is to make a final binary decision whether each point is "typical" or "anomalous". A good way to do this is to look at the approximate distribution of the anomaly scores with the `SArray.sketch_summary` tool, then to get a threshold for the anomaly score with the sketch summary's `quantile` method. Here we declare the top two percent of the data to be anomalies.

```
sketch = scores['anomaly_score'].sketch_summary()
threshold = sketch.quantile(0.98)
anomalies = scores[scores['anomaly_score'] > threshold]
anomalies.print_rows(3, max_row_width=100, max_column_width=20)
```

timestamp	anomaly_score	count	moving_average	model_update_time
2012-12-19 00:00:00	4.2314953527	1020	1880.86666667	2016-01-11 15:41...
2013-05-21 00:00:00	3.7626308199	2481	4535.33333333	2016-01-11 15:41...
2013-05-27 00:00:00	3.91782937621	769	4130.06666667	2016-01-11 15:41...

[13 rows x 5 columns]

It's clear that some of the anomalies are associated with holidays, but others don't have an obvious explanation. It's certainly interesting that all of the anomalous days have *lower* counts than the moving average for those days. For this type of univariate data, it's very useful to plot the anomalies on the original series.



Updating the model with new data

The Moving Z-score is unique among GraphLab Create models in that a new model can be created by updating an existing model. This allows the new model to use the end of the existing model's series to compute moving Z-scores for the first few points in the new data (avoiding the 'None's at the beginning of our first model's output). For this analysis we'll use the 169 observations since June 1, 2015 as "new" data.

```
new_traffic = daily_counts[daily_counts['timestamp'] >= dt.datetime(2015, 06, 01)]
new_model = model.update(new_traffic)
```

Creating a new model with the `update` method *does not change* the original model. The new model looks and smells just like our original model, but there are two small differences in the results. First there are no more missing values in the `scores` `TimeSeries`, because the moving window is filled from the previous model's data.

```
new_scores = new_model['scores']
new_anomalies = new_scores[new_scores['anomaly_score'] > threshold]
new_scores.print_rows(3, max_row_width=100, max_column_width=20)
```

```
+-----+-----+-----+-----+
|      timestamp      | anomaly_score | count | moving_average | model_update_time |
+-----+-----+-----+-----+
| 2015-05-11 00:00:00 | 0.0304807047151 | 3676 | 3650.8 | 2016-01-11 15:41... |
| 2015-05-12 00:00:00 | 0.0769604795962 | 3653 | 3592.13333333 | 2016-01-11 15:41... |
| 2015-05-13 00:00:00 | 1.16177184304 | 2707 | 3619.46666667 | 2016-01-11 15:41... |
+-----+-----+-----+-----+
[169 rows x 5 columns]
```

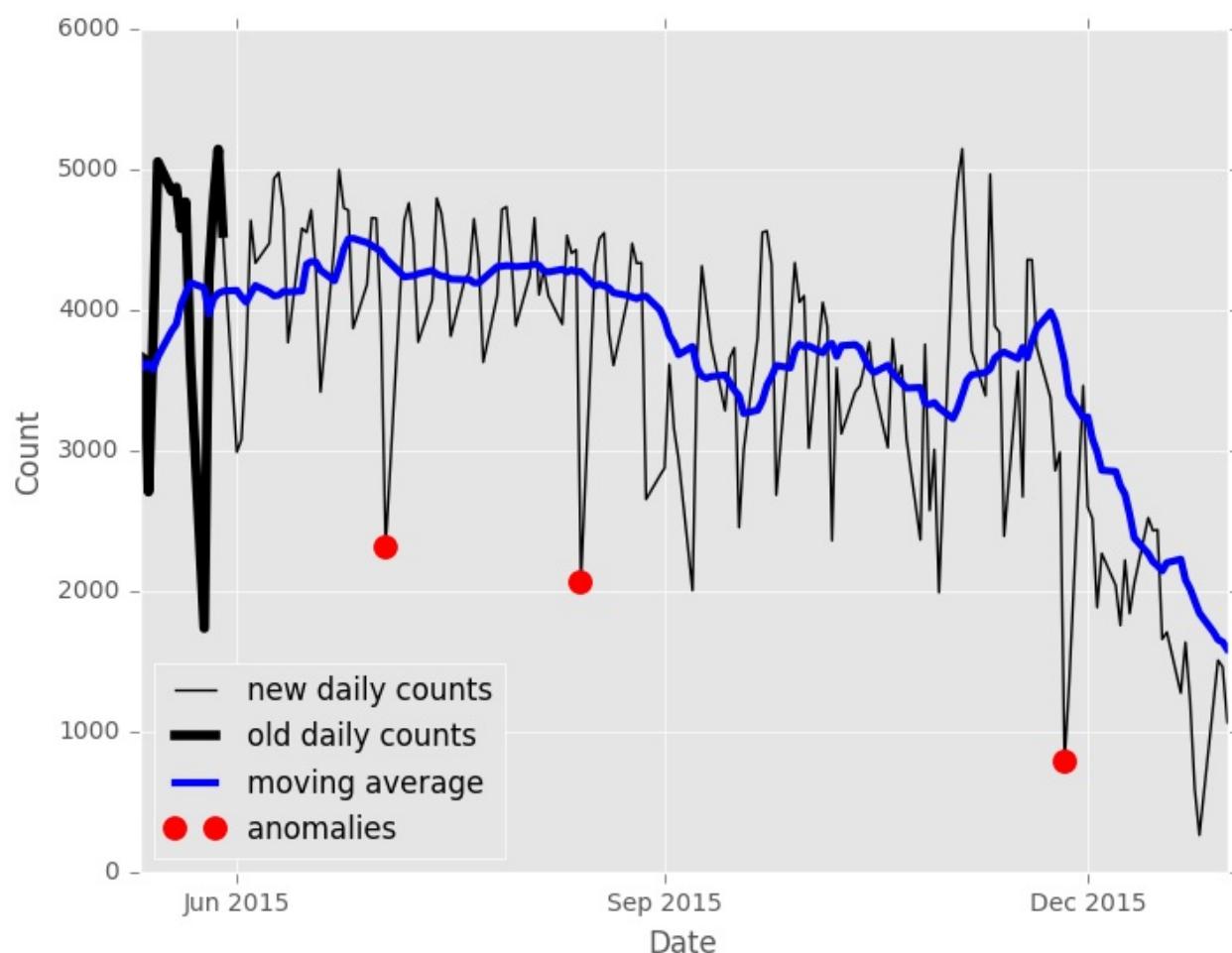
The second difference is that the `model_update_time` is no longer identical for each observation.

```
print new_scores['model_update_time'].unique()
```

```
[datetime.datetime(2016, 1, 11, 15, 41, 1, 777038),
 datetime.datetime(2016, 1, 11, 18, 2, 55, 151691)]
```

The new model's output contains the last `window_size` observations so we can see how the new Z-scores are computed, but these are marked with the original model creation time. We can use this time to separate the scores, so we can plot just the new data's anomalies.

```
update_time = new_scores['model_update_time'][-1]
old_window = new_scores['model_update_time'] < update_time
old_scores = new_scores[old_window]
```



Further reading

Jake Vanderplas wrote [an in-depth analysis](#) of the Fremont bridge bicycle traffic data. The goal of his analysis is not anomaly detection, but it is an excellent read.

Bayesian Changepoints

The [Bayesian Changepoints](#) model scores changepoint probability in a univariate sequential dataset, often a time series. Changepoints are abrupt changes in the mean or variance of a time series. For instance, during an economic recession, stock values might suddenly drop to a very low value. The time at which the stock value dropped is called a changepoint.

Background

The Bayesian Changepoints model is an implementation of the [Bayesian Online Changepoint Detection](#) algorithm developed by Ryan Adams and David MacKay. This algorithm computes a probability distribution over the possible run lengths at each point in the data, where run length refers to the number of observations since the last changepoint. When the probability of a 0-length run spikes, there is most likely a change point at the current data point.

Step 1: Observe new datum x_t and evaluate the likelihood of seeing this value for each possible run length. This is a probability vector, with an element for all possible run lengths. This algorithm assumes a Gaussian distribution between each pair of changepoints.

$$L(r) = P(x|x_r)$$

Step 2: For each possible run length $r > 0$ at current time t , calculate the probability of growth. `expected_runlength` is a parameter describing the a-priori best guess of run length. The larger `expected_runlength` is, the stronger the evidence must be in the data to support a high changepoint probability.

$$P_t(\text{runlength} = r) = P_{t-1}(\text{runlength} = r - 1) * L(r) * (1 - 1/\text{expected_runlength})$$

Step 3: Calculate probability of change, or $r = 0$.

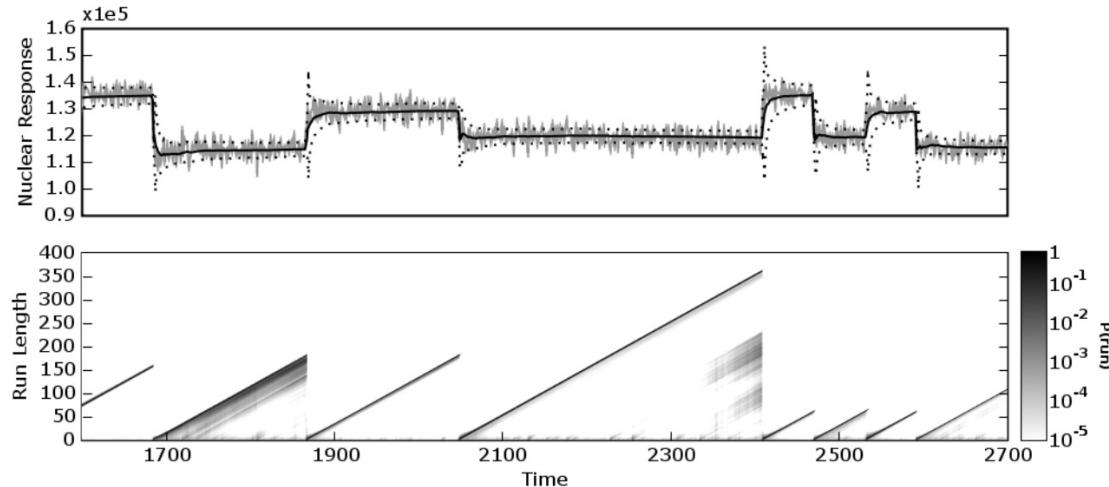
$$P_t(\text{runlength} = 0) = \sum_{r_prev} (P_{t-1}(\text{runlength} = r_prev) * L(0) * (1/\text{expected_runlength}))$$

Step 4: Normalize the probability. For all run length probabilities at time t , divide by the sum of all run length probabilities.

$$P_t(\text{runlength} = r_i) = \frac{P_t(\text{runlength} = r_i)}{\sum_r P_t(\text{runlength} = r)}$$

For each incoming point, repeat this process. This per-point update is why the method is considered an **online** learning algorithm.

The output of this model is a probability distribution of run lengths for each point in the training data. A good example of what this might look like comes from the original [paper] (<https://hips.seas.harvard.edu/files/adams-changepoint-tr-2007.pdf>). This comes from Figure 2.



However, such a distribution is not the desired output of a single changepoint probability for each point. However, the desired output can be obtained by looking at the probability of run length 0 at each point.

As described, the algorithm scores each point x_t immediately, but if the user can afford to wait several observations, it is often more accurate to assign *lagged* changepoint scores. The number of observations to wait before scoring a point is set with the `lag` parameter.

Data and context

The [Dow Jones Industrial Average](#) is an aggregate index of the stock prices of 30 large US companies. Since the stock market goes through ups and downs, and the United States has gone through significant growth and recession, it'd be interesting to see if we can identify when the market has made sudden changes.

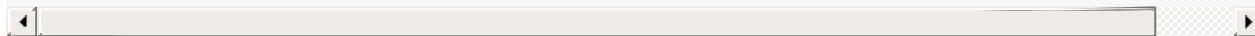
We're interested in the daily percent growth to identify periods of change in the US market. This data, from 2/3/2006 to 2/4/2010, can be downloaded from the website from the [Federal Reserve Bank of St. Louis](#). In order to make things easy, though, we've uploaded the csv file to a public s3 bucket.

We load the file into an SFrame, and convert the values to floats. As a last formatting step, we convert the dataset into a `TimeSeries` by converting the `DATE` column as `datetime.datetime` type and setting that column as the index.

```
# Set up
import graphlab as gl

# Download data.
dow_jones = gl.SFrame.read_csv('https://static.turi.com/datasets/changepoint_djia/DJIA.csv')
dow_jones['VALUE'] = dow_jones['VALUE'].apply(lambda x: None if x == '.' else float(x))
dow_jones['DATE'] = dow_jones['DATE'].str_to_datetime()

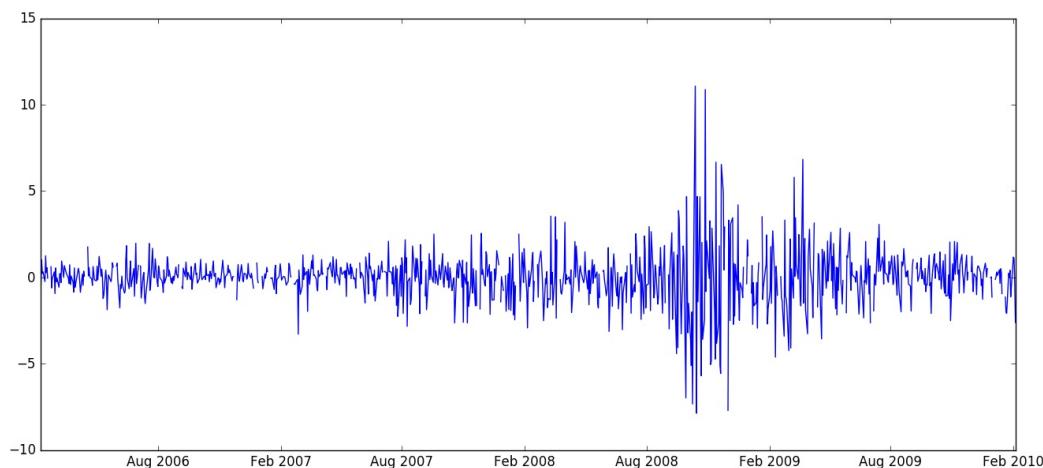
dow_jones = gl.TimeSeries(dow_jones, index='DATE')
```



Our final dataset `dow_jones` has 1045 daily percent returns, from February 3, 2006 to February 4, 2010. You can clearly see the economic downturn of 2008 if you plot it.

```
dow_jones.print_rows(5)
```

```
+-----+-----+
|      DATE      |  VALUE   |
+-----+-----+
| 2006-02-03 00:00:00 |  None    |
| 2006-02-06 00:00:00 |  0.04308 |
| 2006-02-07 00:00:00 | -0.44924 |
| 2006-02-08 00:00:00 |  1.01267 |
| 2006-02-09 00:00:00 |  0.22775 |
+-----+-----+
[1045 rows x 2 columns]
```



Later in this chapter, we'll illustrate how Bayesian Changepoints models can be updated with new data. To do so, we set aside the last 90 days of data.

```
## Split dataset into original and update sets.  
dow_jones_part1 = dow_jones[:-90]  
dow_jones_part2 = dow_jones[90:]
```

Basic Bayesian Changepoints Usage

The Bayesian Changepoints model takes an `SFrame` or `TimeSeries` as input (`dow_jones` in this case), the name of the column that contains the series to model, the number of time points to wait before evaluating the changepoint probability, and the expected number of observations between changepoints. For this analysis our feature is the the "VALUE" column and we use a 20 day lag, since it takes a while to validate a trend in the stock market. We do not really know how often changepoints occur, so we'll leave that value at its default.

```
model = gl.anomaly_detection.bayesian_changepoints.create(dow_jones_part1,  
                                                       feature='VALUE', lag=20)
```

The primary output of the Bayesian Changepoints model is the `scores` field. This `TimeSeries` object contains the original series (`VALUE`), original row index (`DATE`), changepoint score, and the time the model was created (for model updating - see below). For the Bayesian Changepoints model, the changepoint score is the probability of change at that point. This score ranges from 0 to 1, with higher scores indicating a greater probability of being a changepoint.

```
scores = model['scores']  
scores.print_rows(20, max_row_width=100, max_column_width=20)
```

DATE	changepoint_score	VALUE	model_update_time
2006-02-03 00:00:00	None	None	2016-02-04 19:46...
2006-02-06 00:00:00	0.00392332713525	0.04308	2016-02-04 19:46...
2006-02-07 00:00:00	0.0023870906571	-0.44924	2016-02-04 19:46...
2006-02-08 00:00:00	0.00103984848506	1.01267	2016-02-04 19:46...
2006-02-09 00:00:00	0.000342071625611	0.22775	2016-02-04 19:46...
2006-02-10 00:00:00	0.000259090418498	0.32802	2016-02-04 19:46...
2006-02-13 00:00:00	0.000241273784778	-0.2448	2016-02-04 19:46...
2006-02-14 00:00:00	0.000195329199522	1.24923	2016-02-04 19:46...
2006-02-15 00:00:00	0.000281002968276	0.27728	2016-02-04 19:46...
2006-02-16 00:00:00	0.000255319804193	0.55801	2016-02-04 19:46...
2006-02-17 00:00:00	0.000300411758173	-0.0482	2016-02-04 19:46...
2006-02-20 00:00:00	None	None	2016-02-04 19:46...
2006-02-21 00:00:00	None	None	2016-02-04 19:46...
2006-02-22 00:00:00	0.000193007275364	0.61532	2016-02-04 19:46...
2006-02-23 00:00:00	0.000250491249428	-0.61012	2016-02-04 19:46...
2006-02-24 00:00:00	0.00013536712809	-0.06658	2016-02-04 19:46...
2006-02-27 00:00:00	0.000115895087209	0.32273	2016-02-04 19:46...
2006-02-28 00:00:00	0.000139272890529	-0.93841	2016-02-04 19:46...
2006-03-01 00:00:00	9.06744999585e-05	0.54687	2016-02-04 19:46...
2006-03-02 00:00:00	9.61764206427e-05	-0.25349	2016-02-04 19:46...

[955 rows x 4 columns]

Note that values of None result in a changepoint_score of None. Also note that if the input dataset is an `SFrame` instead of a `TimeSeries`, the `scores` field is also an `SFrame`.

One interesting thing is that if you look at the tail of `scores`, you will see a handful of missing values. These data points have insufficient data *after* them to compute lagged changepoint scores. To reduce the number of missing values in the tail of the dataset, reduce the `lag` parameter (at the cost of reducing the accuracy of the results), or update the model with new data (see below).

```
scores.tail(10).print_rows(10, max_row_width=100, max_column_width=20)
```

DATE	changepoint_score	VALUE	model_update_time
2009-09-18 00:00:00	None	0.37081	2016-02-04 21:05...
2009-09-21 00:00:00	None	-0.42097	2016-02-04 21:05...
2009-09-22 00:00:00	None	0.52164	2016-02-04 21:05...
2009-09-23 00:00:00	None	-0.82727	2016-02-04 21:05...
2009-09-24 00:00:00	None	-0.4217	2016-02-04 21:05...
2009-09-25 00:00:00	None	-0.43523	2016-02-04 21:05...
2009-09-28 00:00:00	None	1.28471	2016-02-04 21:05...
2009-09-29 00:00:00	None	-0.48175	2016-02-04 21:05...
2009-09-30 00:00:00	None	-0.30712	2016-02-04 21:05...
2009-10-01 00:00:00	None	-2.09014	2016-02-04 21:05...

[10 rows x 4 columns]

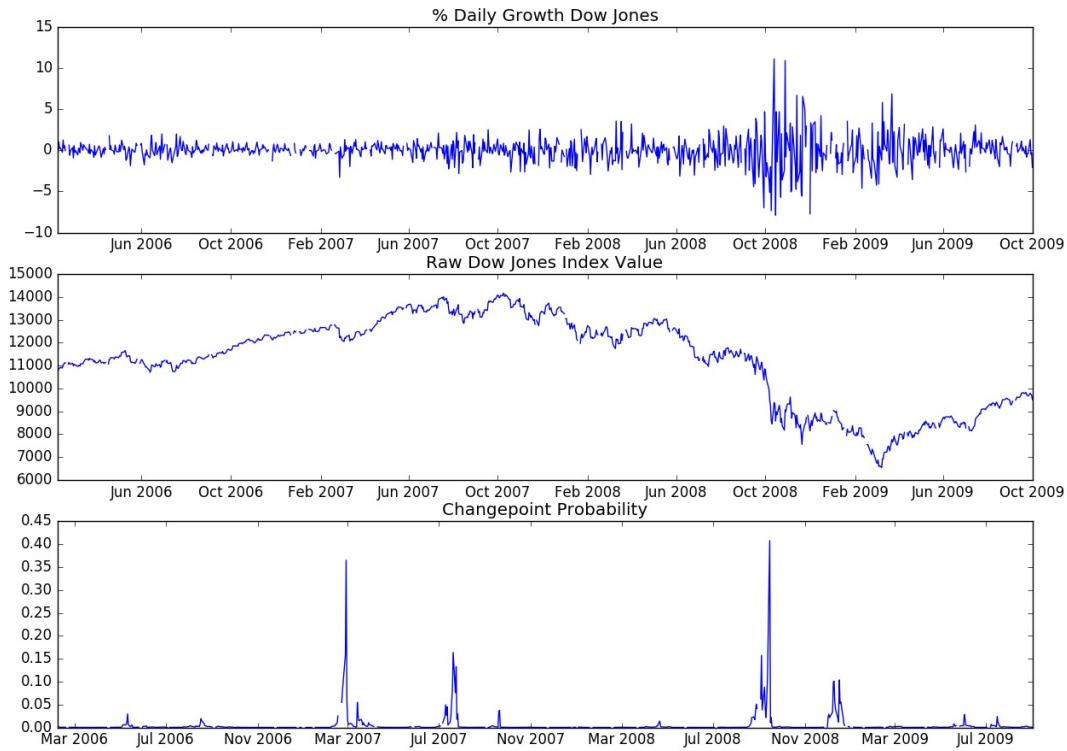
Typically the ultimate goal is to make a final binary decision whether each point is a changepoint or not. A good way to do this is to look at the approximate distribution of the changepoint scores with the `SArray.sketch_summary` tool, then to get a threshold for the changepoint score with the sketch summary's `quantile` method. Here we declare the top 0.5 percent of the data to be changepoints.

```
sketch = scores['changepoint_score'].sketch_summary()
threshold = sketch.quantile(0.995)
changepoints = scores[scores['changepoint_score'] > threshold]
changepoints.print_rows(4, max_row_width=100, max_column_width=20)
```

DATE	changepoint_score	VALUE	model_update_time
2007-02-26 00:00:00	0.158197667441	-0.12034	2016-02-04 21:05...
2007-02-27 00:00:00	0.365058864632	-3.29331	2016-02-04 21:05...
2007-07-20 00:00:00	0.163697924769	-1.06661	2016-02-04 21:05...
2008-09-15 00:00:00	0.40784894056	-4.41674	2016-02-04 21:05...

[4 rows x 4 columns]

Looking at the value of the Dow Jones industrial average in that time(see below), it's clear that high changepoint scores correlate strongly with strong downturns in the market in Februray 2007, end of July 2007, and the huge tumble in the middle of September 2008. In fact, it's interesting that we've identified the exact day that Freddie Mac ceases its purchases of the riskiest subprime mortgages and mortgage-backed securities (2/26/2007) and the day Lehman Brothers file for bankruptcy (09/15/2008).



Percentage growth (our input to the Bayesian Changepoints model), the changepoint probability, and the actual index value for reference.

Updating the model with new data

The Bayesian Changepoints and Moving Z-score models, are unique among GraphLab Create models in that a new model can be created by updating an existing model. This allows the new model to be used in a batch-online fashion.

```
new_model = model.update(dow_jones_part2)
```

Creating a new model with the `update` method *does not change* the original model. There are two things to note about the new model. First the points where we had not waited long enough for `changepoint_probabilities` now have associated probabilities, and those are prepended to the model scores.

```
new_scores = new_model['scores']
new_scores.print_rows(30, max_row_width=100, max_column_width=30)
```

DATE	changepoint_score	VALUE	model_update_time
2009-09-02 00:00:00	0.000674372668387	-0.32146	2016-02-04 21:05:17.547419
2009-09-03 00:00:00	0.000417307023677	0.68896	2016-02-04 21:05:17.547419
2009-09-04 00:00:00	0.000275058596499	1.03439	2016-02-04 21:05:17.547419
2009-09-07 00:00:00	None	None	2016-02-04 21:05:17.547419
2009-09-08 00:00:00	None	None	2016-02-04 21:05:17.547419
2009-09-09 00:00:00	0.000329395813338	0.5252	2016-02-04 21:05:17.547419
2009-09-10 00:00:00	0.000293130519887	0.84066	2016-02-04 21:05:17.547419
2009-09-11 00:00:00	0.000281083172509	-0.22924	2016-02-04 21:05:17.547419
2009-09-14 00:00:00	0.000255184169102	0.22269	2016-02-04 21:05:17.547419
2009-09-15 00:00:00	0.000226709470943	0.58805	2016-02-04 21:05:17.547419
2009-09-16 00:00:00	0.000152394856101	1.11841	2016-02-04 21:05:17.547419
2009-09-17 00:00:00	0.00017838429795	-0.07956	2016-02-04 21:05:17.547419
2009-09-18 00:00:00	0.000152458320239	0.37081	2016-02-04 21:05:17.547419
2009-09-21 00:00:00	0.000127300540065	-0.42097	2016-02-04 21:05:17.547419
2009-09-22 00:00:00	0.000115589044565	0.52164	2016-02-04 21:05:17.547419
2009-09-23 00:00:00	0.000119067795776	-0.82727	2016-02-04 21:05:17.547419
2009-09-24 00:00:00	9.01009242168e-05	-0.4217	2016-02-04 21:05:17.547419
2009-09-25 00:00:00	8.26140126824e-05	-0.43523	2016-02-04 21:05:17.547419
2009-09-28 00:00:00	7.95953975921e-05	1.28471	2016-02-04 21:05:17.547419
2009-09-29 00:00:00	8.78955602003e-05	-0.48175	2016-02-04 21:05:17.547419
2009-09-30 00:00:00	8.61853850655e-05	-0.30712	2016-02-04 21:05:17.547419
2009-10-01 00:00:00	7.9945496499e-05	-2.09014	2016-02-04 21:05:17.547419
2009-10-02 00:00:00	7.6297813879e-05	-0.22725	2016-02-04 21:11:17.881974
2009-10-05 00:00:00	7.902776815e-05	1.18132	2016-02-04 21:11:17.881974
2009-10-06 00:00:00	7.42948152373e-05	1.36983	2016-02-04 21:11:17.881974
2009-10-07 00:00:00	7.62621484995e-05	-0.05827	2016-02-04 21:11:17.881974
2009-10-08 00:00:00	7.94601613024e-05	0.63019	2016-02-04 21:11:17.881974
2009-10-09 00:00:00	8.00705790593e-05	0.7977	2016-02-04 21:11:17.881974
2009-10-12 00:00:00	0.000101576453325	0.21146	2016-02-04 21:11:17.881974
2009-10-13 00:00:00	0.000106263507785	-0.1491	2016-02-04 21:11:17.881974

[112 rows x 4 columns]

The second difference is that the `model_update_time` is no longer identical for each observation.

```
print new_scores['model_update_time'].unique()
```

```
[datetime.datetime(2016, 2, 4, 21, 5, 17, 547419),
 datetime.datetime(2016, 2, 4, 21, 11, 17, 881974)]
```

References

This method is an implementation of [Bayesian Online Changepoint Detection](#) by Ryan Adams and David MacKay so for more in-depth reading please see this paper.

Turi Distributed Introduction

In this chapter, we demonstrate how to use Turi Distributed, a distributed and asynchronous execution framework that makes it easy to take your prototypes to production. Turi Distributed provides a light-weight framework for creating an environment (for example, in EC2 or Hadoop) for distributed execution and submit jobs to these environments, with loose coupling, and management tools to support asynchronous execution.

In order to work with a Turi Distributed deployment (either on an on-premises cluster, or in the Cloud), you will use a GraphLab Create client. The APIs under `graphlab.deploy` provide the necessary functionality to create, use, and administer Turi Distributed environments. Moreover, some of the runtime information about distributed job execution can be visualized in GraphLab Canvas.

Jobs					
Status	Job Name	Job Type	Creation Date	Duration	Type
Completed	my_workflow-May-08-2015-14-33-18	EcdJob	2015-05-08 14:33:18	11m 22 sec	
Completed	my_workflow-May-08-2015-14-29-05	LocalAsynchronousJob	2015-05-08 14:29:05	17m 26 sec	
Failed	my_workflow-May-08-2015-13-41-30	LocalAsynchronousJob	2015-05-08 13:41:30	
Completed	add-May-07-2015-12-48-51	LocalAsynchronousJob	2015-05-07 12:48:51	1 sec	
Completed	add-May-07-2015-12-49-16	LocalAsynchronousJob	2015-05-07 12:49:16	1 sec	
Completed	add-May-07-2015-11-53-10	EcdJob	2015-05-07 11:53:10	6 sec	
Failed	add-May-07-2015-11-19-38	LocalAsynchronousJob	2015-05-07 11:19:38	
Completed	add-May-07-2015-11-19-06	LocalAsynchronousJob	2015-05-07 11:19:06	1 sec	
Completed	sleep_long3	EcdJob	2015-05-04 10:55:54	6m 50 sec	

The following chapters provide more details on the following aspects of remote and asynchronous job execution:

[Asynchronous Jobs](#) describes how you can execute jobs asynchronously, but still within your local machine. Note that this functionality does not depend on Turi Distributed.

[Installing on Hadoop](#) explains how to install Turi Distributed on your local Hadoop environment.

[Clusters](#) provides a walk-through of submitting jobs to EC2 as well as Hadoop.

An [end-to-end example](#) demonstrates how to implement a recommender and run it as a remote job.

[Distributed Machine Learning](#) introduces the concept of executing Turi toolkits in a distributed environment transparently.

[Monitoring Jobs](#) outlines how to gain insight into the status and health of previously submitted jobs.

[Session Management](#) contains information about how to maintain local references to jobs and environments.

The chapter about [Dependencies](#) explains how external packages required by your use case can be included in the job deployment and execution.

Asynchronous Job Executions

This section describes how to execute a job asynchronously, but on the same machine. Consequently this does not count as a remote or distributed execution, and hence does not depend on Turi Distributed. For the sake of completeness it is still included in this chapter.

Let's start with the "Hello World" of deployment examples: adding two numbers. In the following code, we will do the following:

- Write a simple python function to add two numbers.
- Execute the function asynchronously on your local machine.

First, let's create the Python function. Then pass the name and the function keyword arguments that you want to run with into [job.create\(\)](#).

```
import graphlab as gl

def add(x, y):
    return x + y

# Execute the job.
job = gl.deploy.job.create(add, x=1, y=1)
```

Note that the parameter names in the kwargs of the `job.create` call need to match the parameter names in the definition of your method (`x` and `y` in this example).

To get the results of this execution, simply call [job.get_results\(\)](#).

```
print job.get_results()
```

```
2
```

To get the status of this execution, simply call [job.get_status\(\)](#)

```
print job.get_status()
```

```
Completed
```

If the execution of this function throws an exception, we can get the exception type, message, and traceback from the job metrics. See [job.get_metrics\(\)](#).

```
# Will fail since y is None
job = gl.deploy.job.create(add, x=1, y=None)
metrics = job.get_metrics()

print metrics
```

```
+-----+-----+-----+-----+
| task_name | status |      start_time      | run_time | exception |
+-----+-----+-----+-----+
|    add    | Failed | 2015-05-07 11:13:40 |     None   | TypeError  |
+-----+-----+-----+-----+
+-----+
|      exception_message      |      exception_traceback      |
+-----+
| unsupported operand type(s)... | Traceback (most recent call... |
+-----+
[1 rows x 7 columns]
```

```
# get exception type and exception message
print metrics[0]['exception'] + ": " + metrics[0]['exception_message']
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

To visualize this job execution, use [job.show\(\)](#)

```
job.show()
```

That should give you a sense of the types of tasks that can be accomplished with this API. In the following more practical example, we build a recommender and then execute it remotely.

Setting up Turi Distributed on Hadoop

Turi Distributed runs either in the Cloud (using AWS) or on-premises (Hadoop/YARN). If you use it in the Cloud the setup of Turi Distributed happens behind the scenes, all you need is to provide your credentials for AWS. If you decide to use Turi Distributed in your own Hadoop cluster, you will first need to download and install it there.

Prerequisites

We assume that you already downloaded and installed GraphLab Create on a machine that you will use to interact with your Turi Distributed deployment. This can, but doesn't have to be the same machine you used for downloading and installing Turi Distributed. For more information on obtaining and installing GraphLab Create see [Getting Started](#).

You will need the Turi Distributed package as well as a Turi Distributed product key. Both can be obtained on the [installation page on turi.com](#).

You install Turi Distributed from a host that can access the cluster, which usually means to use the Hadoop client tool, together with the appropriate YARN configuration files. After downloading the Turi Distributed package, you unpack it on this machine. Currently Turi Distributed can only be setup from a Mac or Linux machine (not from a Windows machine).

Hadoop

The Hadoop version has to be version 2.6.0 or later. You may setup a Hadoop cluster through:

- The [Apache website](#). We support Hadoop 2.6.0 or later.
- [Cloudera](#), we support CDH 5.0 or later.
- [Hortonworks](#), we support HDP 2.2 or later.

The machine running the Turi Distributed setup needs to be able to access the Hadoop cluster. That means you need to have Java, the Hadoop client, and an appropriate Hadoop configuration in your client machine. Check with your Hadoop administrator to get the Hadoop configuration file.

Deploy

After you downloaded the Turi Distributed package you will need to unpack it:

```
tar xzvf turi-distrib-0.177.tar.gz
```

Deploying the Turi bits to the cluster happens through the script `setup_turi-distributed.sh` that was extracted as part of the package. You will need to provide a destination path (on HDFS) to the script using the `-d` parameter. This path serves as a deployment location that the user refers to when working with the cluster through the Python API. The script connects to the cluster, creates a folder structure in the destination path, and copies the binaries needed for GraphLab Create's distributed runtime there.

Another required parameter is the Turi Distributed product key file, which you obtained as part of the sign-up process (see above). You specify its location through `-k`.

In order to access the cluster, if not already included in your local environment, you might need to tell the script where to find your YARN configuration files using the parameter `-c`:

```
cd turi-distrib-0.177

./setup_turi-distributed.sh -d hdfs://my.cluster:8020/user/name/dd
                            -k ~/Downloads/Turi-Distributed-License.ini
                            -c ~/yarn-conf
```

This deploys Turi Distributed using the YARN config specified in `~/yarn-conf` to the path `hdfs://my.cluster:8020/user/name/dd`, with the GraphLab Create license in `~/Downloads/Turi-Distributed-License.ini`.

Important Parameters

Let's call out a few other parameters that you might have to specify, depending on your cluster environment.

-p <NODE_TMP_DIR> For a typical Turi Distributed use case we recommend at least 10GB of free space for temporary files that are needed locally on nodes during distributed job execution. Usually this means free space in your cluster nodes' `/tmp` folder. You can override this location by specifying the `-p` parameter when calling the setup script. Note that the provided location needs to exist on all nodes of the cluster. Moreover, make sure that YARN has r/w/x access to this location.

-h <HDFS_TMP_DIR> Turi Distributed runs the GraphLab Create engine, which has the ability to spill its large data structures to disk if necessary. The engine will use the default local tmp location, or `NODE_TMP_DIR` if it is set. However, it also has the ability to spill over to a location on HDFS, which you can explicitly specify using the `-h` parameter. If set, this will take precedence over the local tmp location.

Use

Using a Turi Distributed deployment in a Hadoop cluster is described in the following sections. Here is a quick start.

In order to submit jobs to the cluster, you use a cluster object that is based on the Turi Distributed deployment. As an example, let's assume you have used the path

`hdfs://my.cluster.com:8020/user/name/dd` when executing the setup script. You can create the cluster object now as follows:

```
import graphlab as gl

# Create cluster
c = gl.deploy.hadoop_cluster.create(
    name='test-cluster',
    turi_dist_path='hdfs://my.cluster.com:8020/user/name/dd',
    hadoop_conf_dir='~/yarn-config')

def echo(input):
    return input

j = graphlab.deploy.job.create(echo, environment=c, input='hello world!')

j.get_results()
```

This example also assumes that you have a folder `yarn-config` with the YARN configuration files in your home directory.

The cluster object `c` can be used as environment for `job.create` (like in the example above), `map_job.create`, or `distributed model parameter search`.

Clusters in EC2 and Hadoop

GraphLab Create allows for the execution of jobs on the Turi Distributed platform, which runs on EC2 as well as Hadoop YARN clusters. While Turi Distributed is set up automatically on EC2 on-demand, it needs to be deployed on Hadoop.

In this section, we will walk through the concept of a **Cluster** in the GraphLab Create API and how it can be used to execute jobs remotely, either in Hadoop or in EC2.

The Cluster

The GraphLab Create API includes the notion of a **Cluster**, which serves as a logical environment to host the *distributed* execution of jobs (as opposed to the local host environment, which can be [asynchronous](#), but not distributed). GraphLab Create clusters can be created either in EC2 or on Hadoop YARN; while they can equally be used as environments for running jobs, their behavior is slightly different; hence they are represented by two different types: `graphlab.deploy.Ec2Cluster` and `graphlab.deploy.HadoopCluster`. After creating a cluster object once, it can be retrieved at a later time to continue working with an existing cluster. Below we will elaborate on the specifics of each environment.

Creating a Cluster in EC2

In EC2 a cluster is created in two steps: first, a `graphlab.deploy.Ec2Config` object is created, describing the cluster and how to access AWS. The cluster description includes the properties for EC2 instances that are going to be used to form the cluster, like instance type and region, the security group name, etc. Second, the cluster is launched by calling `ec2_cluster.create`. When creating your EC2 cluster, you must also specify a name, and an S3 path where the EC2 cluster maintains its state and logs.

```
import graphlab as gl

# Define your EC2 environment. In this example we use the default settings.
ec2config = gl.deploy.Ec2Config()

ec2 = gl.deploy.ec2_cluster.create(name='my-cluster',
                                   s3_path='s3://my-bucket',
                                   ec2_config=ec2config,
                                   num_hosts=4)
```

At this point you can use the object `ec2` for remote and distributed job execution.

It is important to note that the `create` call will already start the hosts in EC2, so costs will be incurred at that point. They will be shutdown after an idle period, which is 30 minutes by default or set as parameter (in seconds) in the create method. Setting the timeout to a negative value will cause the cluster to run indefinitely or until explicitly stopped. For example, if you wanted to extend the timeout to one hour you would create the cluster like so:

```
ec2 = gl.deploy.ec2_cluster.create(name='my-cluster',
                                    s3_path='s3://my-bucket',
                                    ec2_config=ec2config,
                                    num_hosts=4,
                                    idle_shutdown_timeout=3600)
```

You can retrieve the properties of a cluster by printing the cluster object:

```
print ec2
```

```
S3 State Path: s3://my-bucket
EC2 Config    : [instance_type: m3.xlarge, region: us-west-2, aws_access_key: ABCDEFG]
Num Hosts     : 4
Status        : Running
```

Creating a Cluster in Hadoop

In order to work with a Hadoop cluster, Turi Distributed needs to be set up on the Hadoop nodes. For instructions on how to obtain and install DD please refer to the [Hadoop setup chapter](#)

When you installed Turi Distributed your provided an installation path that you need to refer to when creating a `HadoopCluster` Object through `graphlab.deploy.hadoop_cluster.create` object. Essentially this path is your client-side handle to the Hadoop cluster within the GraphLab Create API. Moreover, when creating your Hadoop cluster object, you must specify a name, which you can later use to retrieve an existing cluster form your workbench. You can also specify `hadoop_conf_dir`, which is the directory of your custom Hadoop configuration path. If `hadoop_conf_dir` is not specified, GraphLab Create uses your default Hadoop configuration path on your machine.

```

import graphlab as gl

# Define your Hadoop environment
td-deployment = 'hdfs://our.cluster.com:8040/user/name/turi-dist-folder'

hd = gl.deploy.hadoop_cluster.create(name='hadoop-cluster',
                                      turi_dist_path=td-deployment)

```

You can retrieve the properties of a cluster by printing the cluster object:

```
print hd
```

```

Hadoop Cluster:
  Name:                  : hadoop-cluster
  Cluster path           : hdfs://our.hadoop-cluster.com:8040/user/name/turi-distribu
  Hadoop conf dir       : /Users/name/yarn-conf

  Number of Containers: : 3
  Container Size (in mb): 4096
  Container num of vcores: 2
  Port range             : 9100 - 9200

  Additional packages    : ['names']

```

(See Section [Dependencies](#) for more information about additional packages.)

Loading an EC2 Cluster

Unlike a `HadoopCluster`, once an `Ec2Cluster` is created, it is physically running in AWS.

This cluster can be loaded at a later time and/or a separate Python session:

```
c = gl.deploy.ec2_cluster.load('s3://my-bucket')
```

Executing Jobs in a Cluster

In order to execute a job in a cluster, you pass the cluster object to the

`graphlab.deploy.job.create` API, independently of whether it is a Hadoop or an EC2 cluster.

While the job is running, the client machine can be shutdown and the job will continue to run.

In the event that the client process terminates, you can reload the job and check its status.

```

def add(x, y):
    return x + y

# c has been created or loaded before
job = gl.deploy.job.create(add, environment=c, x=1, y=2)

```

Note that the parameter names in the kwargs of the `job.create` call need to match the parameter names in the definition of your method (`x` and `y` in this example).

The syntax for getting job status, metrics, and results are the same for all jobs. You can invoke `job.get_status` to get the status, `job.get_metrics` to get job metrics, and `job.get_results` to get job results.

For example, to get the results:

```
print job.get_results()
```

```
2
```

Jobs can be cancelled using `job.cancel`; note that for an EC2 cluster this does not stop the EC2 hosts.

```
job.cancel()
```

For Hadoop-specific job failures (for instance, preemption), you can use the `job.get_error` API.

It is possible that a job succeeds, but tasks inside a job fail. To debug this, use the `job.get_metrics` API.

EC2 Notes

- Once the execution is complete, the idle timeout period will start, after which the EC2 instance(s) started will be terminated. Launching another job will reset the idle timeout period.
- A set of packages to be installed in addition to graphlab and its dependencies can be specified as a list of strings in the `create` call.
- Execution logs will be maintained in S3 (using the `s3_path` parameter in the cluster creation call).

Hadoop Notes

- Job status is also available through normal Hadoop monitoring, as GraphLab Create

submits jobs using a GraphLab YARN application. Logs for executions are available using Yarn logs.

- The location of the logs is available in the job summary, which can be viewed by calling `print job`. You can also use `job.get_log_file_path` to get the location of the logs.
- If you are using Hadoop in Cloudera HA mode, you need to include `conf.cloudera.hdfs` in your CLASSPATH environment variable.

End-to-End Example: Remotely Generate Batch Recommendations

In this example, we demonstrate how to implement a recommender and run it as a remote job. The recommender is implemented as three functions:

1. a data ingestion and cleaning
2. model training
3. generate recommendations

First we will show how to execute this job on the local host.

Local Execution

```
def clean_file(path):
    """
    Takes a CSV file passed in as a param and cleans it into an SFrame.
    In particular, it parses and drops None values.
    """
    import graphlab as gl

    sf = gl.SFrame.read_csv(path, delimiter='\t')
    sf = sf.dropna()
    return sf
```

Next, we train a model from the cleaned data:

```
def train_model(data):
    """
    Takes an SFrame as input and uses it to train a model,
    setting the train and test sets as outputs along with the trained
    model.
    """
    import graphlab as gl

    model = gl.recommender.create(data, user_id='user',
                                  item_id='movie',
                                  target='rating')
    return model
```

Let's make some recommendations based on the model and store them in an SFrame:

```
def recommend_items(model, data):
    recommendations = model.recommend(users=data['user'])
    return recommendations
```

Putting the pieces together:

```
def my_workflow(path):
    # Clean file
    data = clean_file(path)

    # Train model.
    model = train_model(data)

    # Make recommendations.
    recommendations = recommend_items(model, data)

    # Return the SFrame of recommendations.
    return recommendations
```

Having defined the function, we can execute it as a job using the `job.create()` function.

```
job_local = gl.deploy.job.create(my_workflow,
                                 path = 'https://static.turi.com/datasets/movie_ratings/sample.large')

# get status immediately after creating this job.
job_local.get_status()
```

'Running'

Note that we omitted the `environment` parameter, since `LocalAsync` is the default environment when creating jobs.

EC2

Next, let's run our job on EC2. When running on EC2, a cluster defines the EC2 instance to be launched, and is passed to the job for remote execution. After the job is completed and an additional timeout has passed the EC2 instance is terminated. While executing, the job can be monitored with the Job APIs. Execution logs will be stored in S3 according to the location specified in the cluster.

Note: In order to run in EC2, remember to update the `aws_access_key`, `aws_secret_key`, and `s3_path` in the code below.

```

ec2config = gl.deploy.Ec2Config(region='us-west-2',
                                instance_type='m3.xlarge',
                                aws_access_key_id='xxxx',
                                aws_secret_access_key='xxxx')

ec2 = gl.deploy.ec2_cluster.create(name='ec2',
                                    s3_path='s3://bucket/path',
                                    ec2_config=ec2config)

job_ec2 = gl.deploy.job.create(my_workflow,
                               environment=ec2,
                               path='https://static.turi.com/datasets/movie_ratings/sample.large')

# get the results
job_ec2.get_results()

```

The result of this job execution is an `SFrame` containing the recommendations.

```

Columns:
  user      str
  movie     str
  score     float
  rank      int

Rows: 1000000000

Data:
+-----+-----+-----+-----+
|   user |       movie          |   score    | rank |
+-----+-----+-----+-----+
| Jacob Smith | Coral Reef Adventure | 4.28305720509 | 1   |
| Jacob Smith | The Sting           | 3.82596849622 | 2   |
| Jacob Smith | Step Into Liquid     | 3.79010831536 | 3   |
| Jacob Smith | Moonstruck          | 3.76760589303 | 4   |
| Jacob Smith | The Shawshank Redemption: ... | 3.73217236222 | 5   |
| Jacob Smith | Chocolat             | 3.7275472802  | 6   |
| Jacob Smith | Standing in the Shadows of... | 3.72574400128 | 7   |
| Jacob Smith | The Green Mile        | 3.70810352982 | 8   |
| Jacob Smith | Sabrina               | 3.69751512231 | 9   |
| Jacob Smith | The Quiet Man         | 3.6969838065  | 10  |
+-----+-----+-----+-----+
[1000000000 rows x 4 columns]
Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.

```

Hadoop

When defining a Hadoop cluster to use as an environment we specify the directory that contains the YARN configuration files.

Note: The example assumes that you have access to a Hadoop cluster, and that you have a [YARN](#) configuration directory in your home directory.

```
# define the environment, then reuse for subsequent jobs
cdh5 = gl.deploy.hadoop_cluster.create('cdh5',
    turi_dist_path='<path-to-your-turi-distributed-dir>',
    hadoop_conf_dir='~/yarn-conf')

job_hadoop = gl.deploy.job.create(my_workflow,
    environment=cdh5,
    path='https://static.turi.com/datasets/movie_ratings/sample.large')

# get the results
job_hadoop.get_results()
```

Columns:

user	str
movie	str
score	float
rank	int

Rows: 100000000

Data:

user	movie	score	rank
Jacob Smith	Coral Reef Adventure	4.28305720509	1
Jacob Smith	The Sting	3.82596849622	2
Jacob Smith	Step Into Liquid	3.79010831536	3
Jacob Smith	Moonstruck	3.76760589303	4
Jacob Smith	The Shawshank Redemption: ...	3.73217236222	5
Jacob Smith	Chocolat	3.7275472802	6
Jacob Smith	Standing in the Shadows of...	3.72574400128	7
Jacob Smith	The Green Mile	3.70810352982	8
Jacob Smith	Sabrina	3.69751512231	9
Jacob Smith	The Quiet Man	3.6969838065	10

[100000000 rows x 4 columns]

Note: Only the head of the SFrame is printed.

You can use `print_rows(num_rows=m, num_columns=n)` to print more rows and columns.

Launching Distributed Jobs

A core benefit of executing jobs on compute fabrics like EC2 or Hadoop is the ability to scale out and distribute the work across nodes. In this section, we will demonstrate how to launch a distributed execution through the [`map_job`] API, which executes the same function, in parallel, with multiple arguments.

Distributed Execution

A `map_job` is nothing more than a `map` of a function applied to a list of arguments. The result of a `map_job` is a list of results from the execution of the function on each of the arguments.

```
# A map job is equivalent to the following
results = [my_func(**kwargs) for kwargs in parameter_set]
```

In this section, we will show a simple example of executing the `map_job` in a distributed environment.

EC2

To setup a distributed EC2 environment, you will need one or more host in your EC2 cluster.

```

import graphlab as gl

def add(x, y):
    return x + y

ec2config = gl.deploy.Ec2Config()

# Define your EC2 cluster to use 3 hosts (instances)
ec2 = gl.deploy.ec2_cluster.create(name='add_ec2',
                                    s3_path='s3://add_test',
                                    ec2_config=ec2config,
                                    num_hosts=3)

# Execute a map_job.
job = gl.deploy.map_job.create(add, [{"x": 20, "y": 20},
                                      {"x": 10, "y": 10},
                                      {"x": 5, "y": 5}],
                                environment=ec2)

# Get a list of results.
print job.get_map_results()

```

```
[40, 20, 10]
```

In the above EC2 job execution, we distribute the three parameter sets in this job to three different hosts. Each host will run the function with its given parameter set.

If any of the executions failed, we can capture it in the job metrics.

```

# Capture exceptions if the execution failed.
job = gl.deploy.map_job.create(add, [{"x": 20, "y": 20},
                                      {"x": None, "y": 10},
                                      {"x": 5, "y": 5}],
                                environment=ec2)

# Exception captured in metrics if the execution failed.
metrics = job.get_metrics()

print metrics

```

```
+-----+-----+-----+-----+-----+
| task_name | status | start_time | run_time | exception |
+-----+-----+-----+-----+
| add-0-0 | Completed | 2015-05-07 12:46:53 | 6.60419464111e-05 | None |
| add-0-1 | Failed | 2015-05-07 12:46:53 | None | TypeError |
| add-0-2 | Completed | 2015-05-07 12:46:53 | 1.50203704834e-05 | None |
+-----+-----+-----+-----+
+-----+-----+
| exception_message | exception_traceback |
+-----+-----+
| None | None |
| unsupported operand type(s)... | Traceback (most recent call... |
| None | None |
+-----+
[3 rows x 7 columns]
```

```
# Capture partial results of functions that didn't fail.
print job.get_map_results()
```

```
[40, None, 10]
```

Note: In Hadoop `job.get_error()` can provide further diagnosis on failed jobs.

You can process the results of the `map_job` using a combiner function. The combiner is used as follows.

```
def add_combiner(**kwargs):
    return sum(kwargs.values())

# Call map, and then combine all the results using the add_combiner.
job = gl.deploy.map_job.create(add, [{x: 20, y: 20},
                                      {x: 10, y: 10},
                                      {x: 5, y: 5}],
                               environment=ec2,
                               combiner_function=add_combiner)

# get_map_results() would still return [40, 20, 10]
# use get_results() to get result from combiner
print job.get_results()
```

```
70
```

Hadoop

For distributed job execution on Hadoop, you will also need more than one container in your execution environment.

```
import graphlab as gl

def add(x, y):
    return x + y

def add_combiner(**kwargs):
    return sum(kwargs.values())

# Define your Hadoop cluster to use 3 containers
dd-deployment = 'hdfs://our.cluster.com:8040/user/name/turi-dist-folder'

hadoop = gl.deploy.hadoop_cluster.create(name='add_hadoop',
                                         turi_dist_path=dd-deployment,
                                         hadoop_conf_dir='~/yarn-conf',
                                         num_containers=3)

# Execute a map_job.
job = gl.deploy.map_job.create(add, [{"x": 20, "y": 20},
                                       {"x": 10, "y": 10},
                                       {"x": 5, "y": 5}],
                                environment=hadoop,
                                combiner_function=add_combiner)

# get map results
print job.get_map_results()
```

```
[40, 20, 10]
```

```
# get combiner result
job.get_results()
```

```
70
```

In the above Hadoop job execution, we distribute the three parameter sets in this job to three different containers. Each container will run the function with its given parameter set. In the end, we combine the results with a combiner function.

Distributed Machine Learning

Note: The distributed machine learning API has been through significant changes in 2.0 and is not backwards compatible

In the previous chapter we showed how to run jobs in a Turi Distributed cluster. While, Job (or Map Job) give you the benefit of executing arbitrary python code in cluster in a map reduce fashion, it does not support distributing the training of machine learning models.

For a set of GraphLab Create toolkits we have enabled distributed model training in a hadoop cluster. We call this *Distributed Machine Learning* or *DML*. In this section, we will demonstrate how to run distributed machine learning tasks.

The toolkits currently supported to run in a distributed execution environment are:

- [Linear regression](#)
- [Logistic classifier](#)
- [SVM classifier](#)
- [Boosted trees classifier](#)
- [Boosted trees regression](#)
- [Random forest classifier](#)
- [Random forest regression](#)
- [Pagerank](#)
- [Label propagation](#)

Let us look at an example which trains a boosted trees classifier model:

```
import graphlab as gl

# Load data
dataset = 'https://static.turi.com/datasets/xgboost/mushroom.csv'
sf = gl.SFrame(dataset)

# Train model
model = gl.boosted_trees_classifier.create(sf, target='label', max_iterations=12)
```

Hadoop

For distributed machine learning on Hadoop, you will need a cluster object based on a Turi Distributed installation in your Hadoop cluster:

```

>>> c = gl.deploy.hadoop_cluster.create('my-first-hadoop-cluster',
                                         'hdfs://path-to-turi-distributed-installation',
                                         num_containers=4,
                                         container_size=4096,
                                         num_vcores=4)

>>> c.hdfs_tmp_dir = hdfs:///tmp
>>> print c
Hadoop Cluster:
    Name:                  : my-cluster
    Cluster path           : hdfs://path-to-turi-distributed-installation

    Number of Containers: : 4
    Container Size (in mb) : 4096
    Container num of vcores : 4
    Port range             : 9100 - 9200
    Node temp directory    : /tmp
    HDFS temp directory   : hdfs:///tmp

    Additional packages     : None

```



For more information about how to set up a cluster in Hadoop see the chapter on [clusters](#).

The following cluster parameters are critical for successfully running distributed model training:

- `container_size` : Memory limit in MB for each worker. Workers which exceed the memory limit may get killed and result in job failure.
- `Node temp directory` : The local temporary directory to store cache and intermediate files. Make sure this directory has enough disk space.
- `HDFS temp directory` : The hdfs temporary directory to store cache and intermediate files. Make sure this directory has enough disk space and is writable by hadoop user `yarn` .

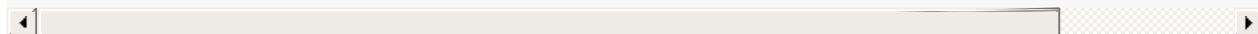
Once the cluster is created you can use it as an execution environment for a machine learning task:

```

sf = gl.SFrame('hdfs://DATASET_PATH')

job = gl.distributed.boosted_trees_classifier.submit_training_job(env=c, dataset=sf, targ

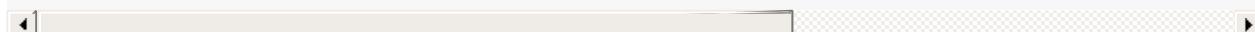
```



The above code submits to the cluster a distributed boosted trees classifier training job which is automatically distributed among the number of containers. The return object is a handle to the submitted job.

```
>>> print job.get_state()
STATE.COMPLETED
```

```
>>> print job.get_progress()
PROGRESS: Number of workers: 4
PROGRESS: CPUs per worker : 4
PROGRESS: Memory limit: 3276.8MB
PROGRESS: Local cache file locations: /tmp
PROGRESS: HDFS access: yes
PROGRESS: HDFS cache file locations: hdfs:///tmp
PROGRESS: Max fileio cache capacity: 819.2MB
PROGRESS: Boosted trees classifier:
PROGRESS: -----
PROGRESS: Number of examples      : 8124
PROGRESS: Number of classes       : 2
PROGRESS: Number of feature columns: 22
PROGRESS: Number of unpacked features : 22
PROGRESS: +-----+-----+-----+
PROGRESS: | Iteration | Elapsed Time | Training-accuracy | Training-log_loss |
PROGRESS: +-----+-----+-----+
PROGRESS: | 1        | 0.124934   | 0.999631      | 0.438946      |
PROGRESS: | 2        | 0.245594   | 0.999631      | 0.298226      |
PROGRESS: | 3        | 0.355051   | 0.999631      | 0.209494      |
PROGRESS: | 4        | 0.489932   | 0.999631      | 0.150114      |
PROGRESS: | 5        | 0.605267   | 0.999631      | 0.109027      |
PROGRESS: Checkpointing to hdfs:///turi_distributed/jobs/dml_job_88a92020-e6d9-42d3-ade3-
PROGRESS: | 6        | 1.255561   | 0.999631      | 0.080002      |
PROGRESS: | 10       | 1.665121   | 0.999631      | 0.024130      |
PROGRESS: Checkpointing to hdfs:///turi_distributed/jobs/dml_job_88a92020-e6d9-42d3-ade3-
PROGRESS: +-----+-----+-----+
```



```
# wait for job to complete
import time
while job.get_state() != job.STATE.COMPLETED:
    time.sleep(1)
    print job.get_progress()
    print job.get_state()

# check the final state
assert job.get_final_state() == job.FINAL_STATE.SUCCESS:

# fetch the trained model, this code will block until job.get_state() is COMPLETED.
model = job.get_results()
model.save('./bst_mushroom')
```

Data locality

Data locality is critical to efficient distributed model training with massive data. In the example above, because the SFrame is constructed from HDFS source, there will be no copy of data between HDFS and the local machine that submits the training job. The model training process will be executed natively in the cluster by reading from HDFS.

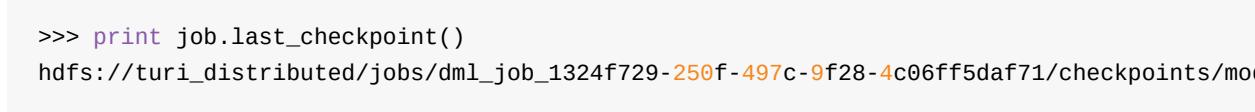
On the other hand, if the SFrame is constructed locally, or read from S3, the SFrame will be automatically copied into HDFS. Depending on the size of the data and network speed, this process can take from minutes to hours. Hence, it is recommended to always save your training or validation SFrame to HDFS before submitting the training job.

Model Checkpointing (Available for Boosted Trees and Random Forest models)

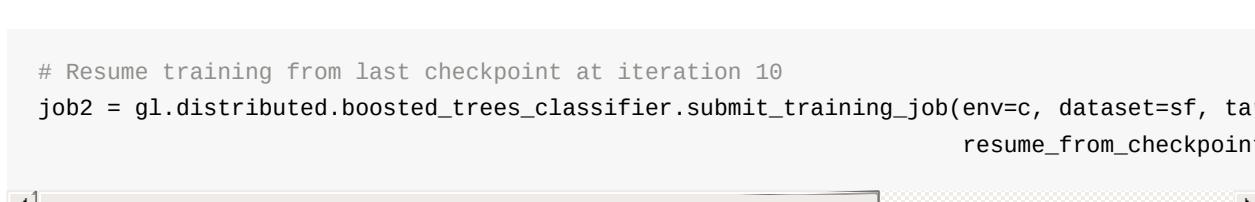
Jobs running in distributed environment like Yarn may be preempted when the load of the cluster is high. Therefore, it is critical to have some recovery mechanism for models that could take very long time to train.

For training a Boosted Trees or Random Forest model, the API supports checkpointing the model every K (default 5) iterations to a file system (local, HDFS or S3) location. In case of interruption, you can resume the training procedure by pointing to the checkpointed model. This feature is enabled by default for distributed boosted trees and random forest model. The training job automatically creates a checkpoint every 5 iterations at the working directory.

```
>>> print job.last_checkpoint()
hdfs://turi_distributed/jobs/dml_job_1324f729-250f-497c-9f28-4c06ff5daf71/checkpoints/mod
```

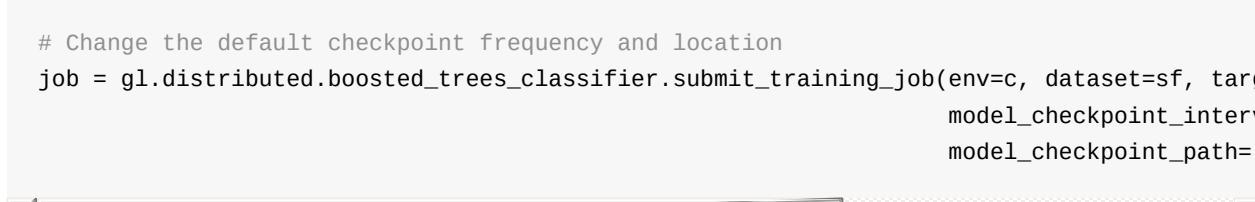


```
# Resume training from last checkpoint at iteration 10
job2 = gl.distributed.boosted_trees_classifier.submit_training_job(env=c, dataset=sf, targ
resume_from_checkpoint
```



You can also specify the location and frequency of checkpointing by using the parameter `model_checkpoint_path` and `model_checkpoint_interval` in `submit_training_job`. For example:

```
# Change the default checkpoint frequency and location
job = gl.distributed.boosted_trees_classifier.submit_training_job(env=c, dataset=sf, targ
model_checkpoint_interv
model_checkpoint_path=
```



Note: The training job is executed as hadoop user `yarn` in the cluster. When specifying `model_checkpoint_path`, please make sure the directory is writable by hadoop user `yarn`.

Debugging Job Failure

The final job state could be one of the followings: `FINAL_STATE.SUCCESS` , `FINAL_STATE.KILLED` , `FINAL_STATE.FAILURE` . When the job does not complete successfully, the following sequence may be executed to debug the failure.

```
>>> print job.get_final_state()
'FINAL_STATE.FAILURE'
>>> print job.summary()
Container list: ['container_e44_1467403925081_0044_01_000001', 'container_e44_1467403925081_0044_01_000002'

container_e44_1467403925081_0044_01_000001:
    Application Master
container_e44_1467403925081_0044_01_000002:
    Report not found in log.
container_e44_1467403925081_0044_01_000003:
    Report not found in log.
container_e44_1467403925081_0044_01_000004:
    Container terminated due to exceeding allocated physical memory. (-104)
```

```
# Get the location of yarn logs, you can open the log file in your local editor.
>>> print job.get_log_file_path()
/tmp/tmpdcl2_V/tmp_log.stdout
```

Monitoring Jobs

Jobs are designed to be asynchronous objects. Once they are created (using the `create()` function), you can manage their execution status using the `get_status()` and `cancel()` methods.

Status

To get the current status of a job:

```
job.get_status()
```

```
'Running'
```

`job.get_status()` will return one of the following messages: 'Pending', 'Running', 'Completed', 'Failed', 'Unknown', 'Canceled'.

Start and End time

To get the start time of this job:

```
job.get_start_time()
```

```
datetime.datetime(2015, 5, 8, 14, 36, 33)
```

To get the end time of this job:

```
job.get_end_time()
```

```
datetime.datetime(2015, 5, 8, 14, 47, 55)
```

Both `job.get_start_time()` and `job.get_end_time()` return a Python `DateTime` object.

Metrics

To get more information about this job, such as why this job failed, or why one of the executions for a set of parameters failed, we look at the job metrics.

```
print job.get_metrics()
```

```
+-----+-----+-----+-----+
| task_name | status | start_time | run_time | exception |
+-----+-----+-----+-----+
| my_workflow | Completed | 2015-05-08 14:36:38 | 650.040112972 | None |
+-----+-----+-----+-----+
+-----+
| exception_message | exception_traceback |
+-----+-----+
| None | None |
+-----+
[1 rows x 7 columns]
```

The metrics [SFrame](#) contains the status, start time, and run time of each execution. We can also find out the exception type, message, and traceback if the execution fails.

Another way to retrieve more details about job execution errors in Hadoop is

```
job.get_error() :
```

```
job.get_error()
```

```
'Failed to install user-specified package.'
```

Logs

To get the log file path for this execution, simply call [job.get_log_file_path\(\)](#). This works for both EC2 (S3 path) and Hadoop (HDFS path).

```
job.get_log_file_path()
```

```
's3://my-bucket/my_workflow-May-08-2015-14-33-16-f6dff07e-2f4a-4cdf-97c9-3d8729bfca6f/log'
```

Cancel

You can cancel a job by calling [job.cancel\(\)](#).

```
job.cancel()
```

Results

To get results of a job execution, we can use `job.get_results()`.

For distributed jobs, we can use `job.get_map_results()` to get the result of each execution. If the distributed job has a combiner function, we can obtain the combiner function result from `job.get_results()`.

Visualize Job

To visualize a Job, call `show()` method:

```
job.show()
```

Status	Parameters	Duration
Completed	path: https://s3.amazonaws.com/turi-datasets/movie_ratings/sample.large	10m 50.04 sec

```
def my_workflow(path):
    # Clean file
    data = clean_file(path)

    # Train model.
    model = train_model(data)

    # Make recommendations.
    recommendations = recommend_items(model, data)

    # Return the SFrame of recommendations.
    return recommendations
```

More

To get more information about a Job, simply print it. This provides general information about the job, including what parameters are being used, the name of the job, and execution-related information will be printed.

For example,

```
print job
```

Info

Job : my_workflow-May-08-2015-14-33-16
Environment : EC2: ["name": ec2, "access_key": ..., "instance_type": m3.xlarge, "r
Function(s) : ['my_workflow']
Status : Completed

Help

Visualize progress : self.show()
Query status : self.get_status()
Get results : self.get_results()

Metrics

Start time : 2015-05-08 14:36:33
End time : 2015-05-08 14:47:55

task_name	status	start_time	run_time	exception	
my_workflow	Completed	2015-05-08 14:36:38	650.040112972	None	
exception_message		exception_traceback			
None		None			

[1 rows x 7 columns]

Execution Information:

EC2 Environment name: ec2
S3 Folder: s3://my-bucket

Session Management

GraphLab Create manages local references to your Jobs and Environments in a local session. These local references can be persisted to disk, allowing you to resume and modify your work at a later time. Put another way, GraphLab Create makes it easy to recall previous work, maintaining a history and a workbench that facilitate incremental modifications over time.

Because these objects are persisted they must have unique names, so the session can keep track of them.

List

To see a listing of all jobs:

```
gl.deploy.jobs
```

```
Job(s):
+-----+-----+-----+-----+
| Index | Environment | Name           | Creation date   |
+-----+-----+-----+-----+
| 0     | my-bucket    | my_workflow-May-08-2015-14... | 2015-05-08 14:36:20+00:00 |
| 1     | async         | my_workflow-May-08-2015-14... | 2015-05-08 14:25:05+00:00 |
| 2     | async         | my_workflow-May-08-2015-13... | 2015-05-08 13:41:30+00:00 |
| 3     | async         | add-May-07-2015-12-46-51    | 2015-05-07 12:46:51+00:00 |
| 4     | async         | add-May-07-2015-12-46-16    | 2015-05-07 12:46:16+00:00 |
| 5     | my-bucket    | add-May-07-2015-11-53-10  | 2015-05-07 11:56:14+00:00 |
| 6     | async         | add-May-07-2015-11-13-38  | 2015-05-07 11:13:38+00:00 |
| 7     | async         | add-May-07-2015-11-13-06  | 2015-05-07 11:13:07+00:00 |
| 8     | my-bucket    | sleep_long3                | 2015-05-04 10:55:42+00:00 |
+-----+-----+-----+-----+
[9 rows x 4 columns]
```

To visualize all Jobs, run:

```
gl.deploy.jobs.show()
```

Status	Job Name	Job Type	Creation Date	Duration
✓ Completed	my_workflow-May-08-2015-14-33-16	Ec2Job	2015-05-08 14:36:33	11m 22 sec
✓ Completed	my_workflow-May-08-2015-14-25-05	LocalAsynchronousJob	2015-05-08 14:25:06	17m 26 sec
✗ Failed	my_workflow-May-08-2015-13-41-30	LocalAsynchronousJob	2015-05-08 13:41:32	-----
✓ Completed	add-May-07-2015-12-46-51	LocalAsynchronousJob	2015-05-07 12:46:52	1 sec
✓ Completed	add-May-07-2015-12-46-16	LocalAsynchronousJob	2015-05-07 12:46:17	1 sec
✓ Completed	add-May-07-2015-11-53-10	Ec2Job	2015-05-07 11:56:26	6 sec
✗ Failed	add-May-07-2015-11-13-38	LocalAsynchronousJob	2015-05-07 11:13:38	-----
✓ Completed	add-May-07-2015-11-13-06	LocalAsynchronousJob	2015-05-07 11:13:08	1 sec
✓ Completed	sleep_long3	Ec2Job	2015-05-04 10:55:54	6m 50 sec

This will show the overall dashboard for all Jobs known in the workbench.

To see all Environments known in the workbench.

```
gl.deploy.environments
```

Environment(s):

Index	Name	Type	Unsaved changes?	Creation date
0	my-bucket	EC2	No	2015-05-07 11:52:34+00:00
1	async	LocalAsync	No	2015-05-07 11:13:07+00:00

[2 rows x 5 columns]

For programmatic access:

```
gl.deploy.jobs.list() # returns an SFrame of jobs
gl.deploy.predictive_services.list() # returns an SFrame of predictive services
gl.deploy.environments.list() # returns an SFrame of environments
```

Load

We can use either the index of the job in the session, or the name, to load a persisted job in the current session.

```
print gl.deploy.jobs
```

Index	Environment	Name	Creation date
0	my-bucket	my_workflow-May-08-2015-14...	2015-05-08 14:36:20+00:00
1	async	my_workflow-May-08-2015-14...	2015-05-08 14:25:05+00:00
2	async	my_workflow-May-08-2015-13...	2015-05-08 13:41:30+00:00
3	async	add-May-07-2015-12-46-51	2015-05-07 12:46:51+00:00
4	async	add-May-07-2015-12-46-16	2015-05-07 12:46:16+00:00
5	my-bucket	add-May-07-2015-11-53-10	2015-05-07 11:56:14+00:00
6	async	add-May-07-2015-11-13-38	2015-05-07 11:13:38+00:00
7	async	add-May-07-2015-11-13-06	2015-05-07 11:13:07+00:00
8	my-bucket	sleep_long3	2015-05-04 10:55:42+00:00

[9 rows x 4 columns]

To load the first job at index "0":

```
job = gl.deploy.jobs[0]
```

To load the job named "sleep_long3":

```
job = gl.deploy.jobs['sleep_long3']
```

Delete

We can also use the index or the name to delete a job from the current session.

```
del gl.deploy.jobs[0]
# or
gl.deploy.jobs.delete[0]
```

```
del gl.deploy.jobs['sleep_long3']
# or
gl.deploy.jobs.delete['sleep_long3']
```

Managing Dependencies

Any additional Python packages required for your job execution need to be known to the framework in order to ensure those packages are installed prior to running the function within a job. Such packages need to be specified when defining the cluster environment through `graphlab.deploy.hadoop_cluster.create` :

```
import graphlab as gl

c = gl.deploy.hadoop_cluster.create(
    name='hadoop-cluster',
    turi_dist_path=<HDFS path to turi distributed deployment>,
    additional_packages='names==0.3.0')

def my_function(number = 10):
    import names
    people = [names.get_full_name() for i in range(number)]
    sf = graphlab.SFrame({'names':people})
    return sf

job = gl.deploy.job.create(my_function, environment=c, number=20)
```

The `additional_packages` parameter can be a single string or a list of strings, describing packages in the pypi format. Equivalent to Hadoop it can also be provided to `graphlab.deploy.ec2_cluster.create` for EC2 clusters. In the case of Hadoop, these packages need to be explicitly uploaded to the cluster (see below). Note that creating a cluster in Hadoop and specifying packages that have not been uploaded will succeed, but a subsequent submission of a job will fail (because Turi Distributed will try to install the specified packages into the job's environment at that point).

Alternatively, dependent packages can be specified on an already created cluster, by setting the cluster's `additional_packages` property, before submitting a job:

```
# load a previously created EC2 cluster:
ec2c = gl.deploy.ec2_cluster.load('s3://my-workspace/ec2-cluster')

ec2c.additional_packages = 'names'
```

Hadoop Package Management

Because Hadoop clusters are not generally connected to the internet, Turi Distributed provides functionality to manage packages in such an environment, namely uploading, listing, and removing packages. These operations are accessible either through the command

line (scripts provided by Turi Distributed) or the GraphLab Create API.

Packages available for installation (as described above) are shown as follows:

```
gl.deploy.hadoop_cluster.show_available_packages(  
    turi_dist_path='hdfs://my.cluster.com:8040/user/name/my-cluster-setup')
```

(In this example we assume (i) that the YARN config folder to access the cluster has been set as default, and (ii) that a Turi Distributed deployment in a folder `my-cluster-setup` in the user's home directory in the cluster exists.)

Output:

```
{'default_packages': ['_license==1.1',  
    'abstract-rendering==0.5.1',  
    'anaconda==2.2.0',  
    ...  
    'yaml==0.1.4',  
    'zeromq==4.0.4',  
    'zlib==1.2.8'],  
    'user_packages': ['names-0.3.0.tar.gz']}
```

A package can be uploaded through the following API:

```
gl.deploy.hadoop_cluster.upload_packages(  
    turi_dist_path=<hdfs path to turi distributed deployment>,  
    filename_or_dir='./names-0.3.0.tar.gz')
```

Alternatively to a single package filename, a folder can be provided; all files within that folder will be uploaded to the cluster, assuming they are Python packages.

Packages can be removed from the cluster as follows:

```
gl.deploy.hadoop_cluster.remove_package(  
    turi_dist_path=<hdfs path to turi distributed deployment>,  
    filename='names-0.3.0.tar.gz')
```

Turi Predictive Services

Turi Predictive Services enable straightforward deployment of machine learning models as reliable and scalable web services for easy integration into predictive applications using a low-latency REST API. With one command, launch a cluster, deploy a model, update a model (with no downtime), or capture interaction data to improve the next version of the model.

As of Predictive Services version 2.0 we have separated its documentation into the standalone [Turi Predictive Services User Guide](#). Please continue to read there and update your bookmarks accordingly.

You can find the previous Predictive Services user guide [here](#).

Conclusion

We hope you have enjoyed learning about how to use GraphLab Create. We have additional resources that you may be interested in:

- [API Reference](#): Find details on how to use particular methods and classes
- [API Translator](#): A reference for translating between GraphLab Create and related pieces of software
- [How-tos](#): A collection of code-snippets and examples
- [Gallery](#): Jupyter notebooks showing end-to-end applications using GraphLab Create

Exercises

As you learn how to use the Turi Machine Learning Platform, try out these exercises to help you test your knowledge.

- [Tabular data](#)
- [Graph data](#)
- [Graph analytics](#)
- [Classification](#)
- [Text analysis](#)
- [Recommender systems](#)

Exercises

The following hands-on exercises will help you learn how to work with data using GraphLab Create.

For these exercises we will use a StackOverflow dataset, which can be obtained as follows as a prepackaged SFrame.

```
# Downloads the data from S3 if you haven't already.
import os
if os.path.exists('stack_overflow'):
    sf = graphlab.SFrame('stack_overflow')
else:
    sf = graphlab.SFrame('https://static.turi.com/datasets/stack_overflow')
    sf.save('stack_overflow')
```

Question 1: Unfortunately, there are lots of missing values in this data that weren't given when parsing. Can you discover what these values are and replace them so GraphLab Create recognizes them as missing? Hint: Some columns are not the type they should be. Use `astype` to cast to the type you need.

```
for i in sf.column_names():
    sf[i] = sf[i].apply(lambda x: None if x == '' else x)
```

Question 2: Come up with criteria for a "double post" and filter out duplicates. How many duplicates did you filter? If predicting if a question will be closed, would you filter these posts out?

```
# This is just one of many definitions of a "double post".  If I was to predict
# whether a question will be closed, I would not filter these out.
tmp = sf.groupby(['Title', 'OwnerUserId', 'BodyMarkdown'], {'PostId':gl.aggregate.SELECT_O
filtered = sf.filter_by(tmp['PostId'], 'PostId')
print len(sf)-len(filtered)
```

Question 3: Convert all of the "tag" columns to one column with a list of tags and called 'tags'.

```
sf = sf.pack_columns(['Tag1', 'Tag2', 'Tag3', 'Tag4', 'Tag5'], new_column_name='tags')
```

Question 4: Display the top 10 question askers.

```
sf.groupby(['OwnerUserId'], {'post_count':gl.aggregate.COUNT()}).topk('post_count', k=10)
```

Columns:

OwnerUserId	str
post_count	int

Rows: 10

Data:

OwnerUserId	post_count
39677	1781
4653	1437
34537	1423
179736	1234
149080	1190
117700	1172
84201	1164
434051	1093
325418	995
4639	974

[10 rows x 2 columns]

Question 5: Find the highest rated asker for the 'not constructive' question type.

```
sf['ReputationAtPostCreation'] = sf['ReputationAtPostCreation'].astype(int)
notc = sf[sf['OpenStatus'] == 'not constructive']
notc.topk('ReputationAtPostCreation', k=1)
```

Columns:

```
PostId      str
PostCreationDate    str
OwnerUserId    str
OwnerCreationDate    str
ReputationAtPostCreation    int
OwnerUndeletedAnswerCountAtPostTime    str
Title      str
BodyMarkdown    str
PostClosedDate    str
OpenStatus    str
tags      list
```

Rows: 1

Data:

PostId	PostCreationDate	OwnerUserId	OwnerCreationDate	ReputationAtPostCr
11058483	06/15/2012 21:32:26	16417	09/17/2008 17:02:31	98265
OwnerUndeletedAnswerCountA	Title			
1000	PHP - Justify string algorithm			
BodyMarkdown	PostClosedDate	OpenStatus		
Just tanked a job interview ...	06/22/2012 13:01:37	not constructive		
tags				
['php', 'algorithm', None, ...]				

[1 rows x 11 columns]

Creating an SGraph

For the following SGraph exercises, we'll use the countries and results from two groups in the 2014 World Cup to explore SGraph construction, summarization, visualization, and filtering. To model this kind of data as a graph, we'll represent countries by vertices and game outcomes by directed edges. The source vertex for each edge is the losing team, and the destination vertex is the winning team. Ties are represented by including edges in both directions.

The following data was transcribed from the [FIFA website](#).

```
countries = {
    'usa': 'G',
    'ghana': 'G',
    'germany': 'G',
    'portugal': 'G',
    'england': 'D',
    'italy': 'D',
    'costa rica': 'D',
    'uruguay': 'D'}

results = [
    ('portugal', 'germany'),
    ('ghana', 'usa'),
    ('portugal', 'usa'),
    ('usa', 'portugal'),
    ('usa', 'germany'),
    ('ghana', 'portugal'),
    ('ghana', 'germany'),
    ('germany', 'ghana'),
    ('england', 'italy'),
    ('england', 'uruguay'),
    ('england', 'costa rica'),
    ('costa rica', 'england'),
    ('uruguay', 'costa rica'),
    ('italy', 'costa rica'),
    ('italy', 'uruguay')]
```

Question 1:

Construct [SFrames](#) for the vertex and edge data.

```
verts = graphlab.SFrame({'name': countries.keys(),
                        'group': countries.values()})
print verts
```

```
+-----+-----+
| group |      name   |
+-----+-----+
|   G   |    ghana   |
|   G   |    germany  |
|   G   |    portugal |
|   G   |     usa     |
|   D   |    uruguay  |
|   D   |    england  |
|   D   |    italy    |
|   D   |    costa rica|
+-----+-----+
[8 rows x 2 columns]
```

```
losers, winners = zip(*results)
edges = graphlab.SFrame({'loser': list(losers),
                        'winner': list(winners)})
print edges
```

```
+-----+-----+
| loser  |  winner |
+-----+-----+
| portugal | germany |
| ghana   |   usa   |
| portugal |   usa   |
| usa     | portugal |
| usa     | germany  |
| ghana   | portugal |
| ghana   | germany  |
| germany | ghana   |
| england | italy   |
| england | uruguay |
| ...     |   ...   |
+-----+-----+
[15 rows x 2 columns]
```

Question 2: Make an empty SGraph.

```
sg = graphlab.SGraph()
```

Question 3: Add the vertices and edges to the graph.

```
sg = sg.add_vertices(verts, vid_field='name')
sg = sg.add_edges(edges, src_field='loser', dst_field='winner')
```

Summarize and visualize the graph

Question 4: Summarize the graph. How many vertices and edges are there?

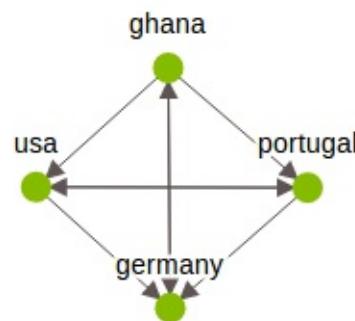
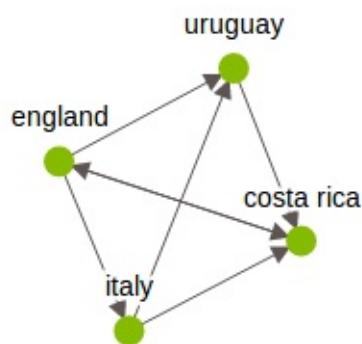
The [summary](#) method gives the number of vertices and edges in the graph, which is often the best place to start.

```
print sg.summary()
```

```
{'num_edges': 15, 'num_vertices': 8}
```

Question 5: Show the graph. Highlight the teams from North America, and use arrows to indicate the winner of each match.

```
sg.show(arrows=True, vlabel='id')
```



Question 6: Extract the vertices and edges as SFrames. Do the numbers of rows and edges match the summary?

The graph's SFrames for vertices and edges show there are indeed 8 vertices (with a 'group' attribute) and 15 edges (with no attributes).

```
sf_vert = sg.vertices
print sf_vert
```

```
+-----+-----+
|   __id    | group |
+-----+-----+
| ghana    | G   |
| costa rica | D   |
| portugal  | G   |
| usa       | G   |
| england   | D   |
| germany   | G   |
| italy     | D   |
| uruguay   | D   |
+-----+-----+
[8 rows x 2 columns]
```

```
sf_edge = sg.edges
print sf_edge
```

```
+-----+-----+
| __src_id | __dst_id |
+-----+-----+
| ghana    | portugal |
| ghana    | usa      |
| ghana    | germany  |
| costa rica | england |
| portugal  | usa      |
| portugal  | germany  |
| usa      | portugal |
| usa      | germany  |
| germany  | ghana    |
| england   | costa rica|
| ...      | ...      |
+-----+-----+
[15 rows x 2 columns]
```

Filter the vertices

Question 7: Extract only the vertices in group H and plot the subgraph.

There are a couple ways to do this. The `get_vertices` command is the most straightforward, but filtering on the underlying SFrame is more flexible.

```
sub_verts = sg.vertices[sg.vertices['group'] == 'G'] # option 1
sub_verts = sg.get_vertices(fields={'group': 'G'}) # option 2
print sub_verts
```

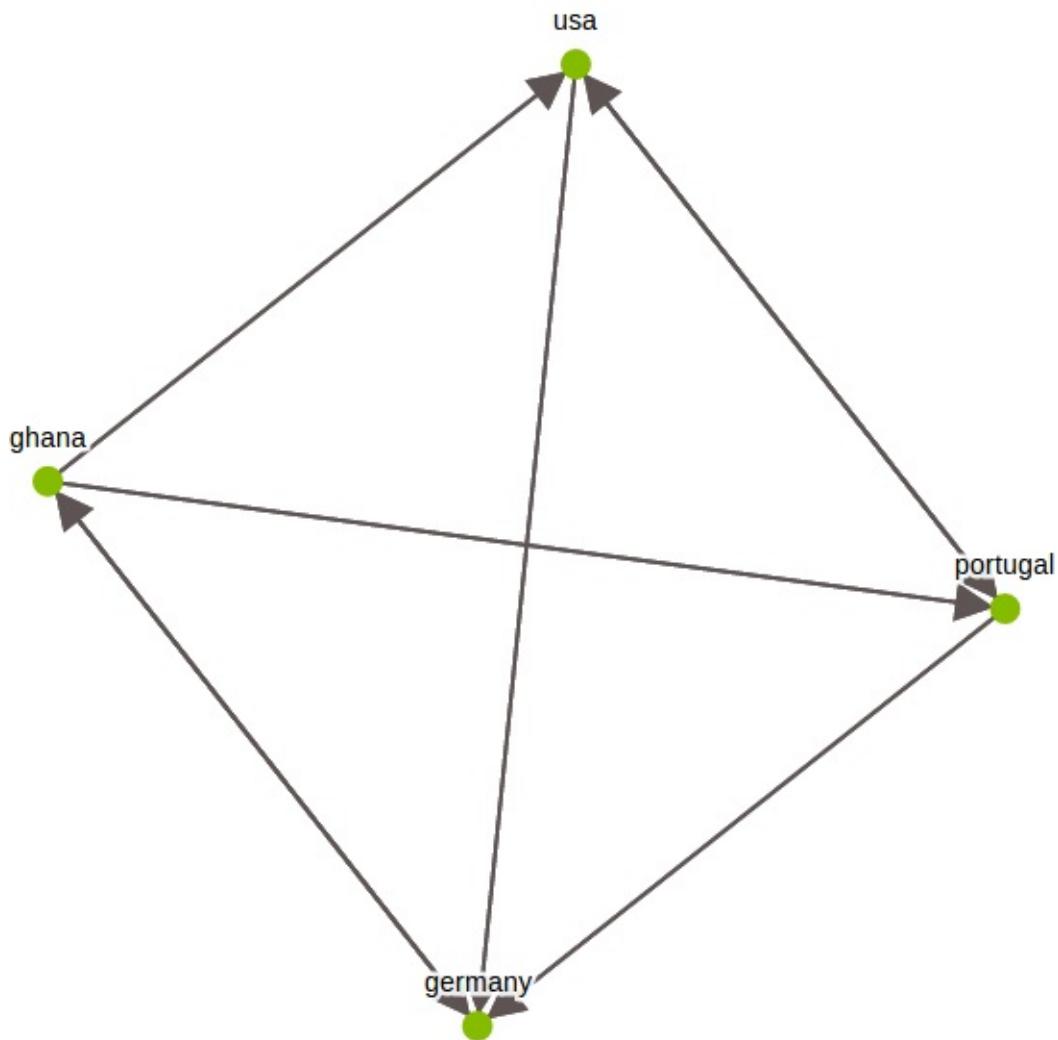
```
+-----+-----+
| __id    | group |
+-----+-----+
| ghana   | G   |
| portugal | G   |
| usa     | G   |
| germany | G   |
+-----+-----+
[4 rows x 2 columns]
```

To materialize the subgraph, construct a new SGraph and add the edges of subgraph.

```
subgraph = graphlab.SGraph()
subgraph = subgraph.add_edges(sg.get_edges(src_ids=sub_verts['__id']))
print subgraph.summary()
```

```
{'num_edges': 8, 'num_vertices': 4}
```

```
subgraph.show(vlabel='id', arrows=True, ewidth=2)
```



Compute vertex degree

The *in-degree* of a vertex is the number of edges that point to the vertex, the *out-degree* is the number of edges that point out from the vertex, and the *degree* is the sum of these two. In the context of our World Cup example, the in-degree is the number of wins and ties, and the degree is the total number of games for a given team. In this set of exercises, we'll use triple apply to compute the in-degree of each vertex. For more on vertex degree, check out the [Wikipedia article](#).

Question 8: Define a function to increment the degree counts for an arbitrary source-edge-destination triple. This function should simply add 1 to the *degree* field for each of the source and destination vertices.

```

def increment_in_degree(src, edge, dst):
    dst['in_degree'] += 1
    return (src, edge, dst)
  
```

Question 9: Create a new vertex field with in-degree set to 0 for each vertex.

Add the 'degree' field to the vertex attributes. Note that adding vertices with the same id's to the graph does not cause duplicate entries.

```
sf_vert['in_degree'] = 0
sg = sg.add_vertices(sf_vert)
```

Question 10: Use the triple apply function to compute in-degree for all nodes. Which team(s) did the best in the group stage?

```
sg = sg.triple_apply(increment_in_degree, mutated_fields=['in_degree'])
print sg.vertices.sort('in_degree', ascending=False)
```

__id	group	in_degree
costa rica	D	3
germany	G	3
portugal	G	2
usa	G	2
uruguay	D	2
ghana	G	1
england	D	1
italy	D	1

[8 rows x 3 columns]

Costa Rica and Germany did the best of these two groups in the group stage. If this is surprising, don't forget that we counted tie games twice.

Find the connected components

In the SGraph exercises in chapter 1 we analyzed results from the group stage of the 2014 World Cup. Now find the connected components for the graph of 2014 World Cup knockout stage results. The data---again from the FIFA website---are listed below. As with the group stage, vertices represent teams and edges point from the losing team to the winning team.

Question 1: First summarize and visualize the new graph.

```

countries = {
    'argentina': 'south america',
    'switzerland': 'europe' ,
    'france': 'europe',
    'greece': 'europe',
    'netherlands': 'europe',
    'costa rica': 'north america',
    'algeria': 'africa',
    'belgium': 'europe',
    'brazil': 'south america',
    'colombia': 'south america',
    'nigeria': 'africa',
    'usa': 'north america',
    'germany': 'europe',
    'chile': 'south america',
    'mexico': 'north america',
    'uruguay': 'south america'}

results = [
    ('chile', 'brazil', 'belo horizonte'),
    ('uruguay', 'colombia', 'rio de janeiro'),
    ('nigeria', 'france', 'brasilia'),
    ('algeria', 'germany', 'porto alegre'),
    ('mexico', 'netherlands', 'fortaleza'),
    ('greece', 'costa rica', 'recife'),
    ('switzerland', 'argentina', 'sao paulo'),
    ('usa', 'belgium', 'salvador'),
    ('colombia', 'brazil', 'fortaleza'),
    ('france', 'germany', 'rio de janeiro'),
    ('costa rica', 'netherlands', 'salvador'),
    ('belgium', 'argentina', 'brasilia'),
    ('brazil', 'germany', 'belo horizonte'),
    ('netherlands', 'argentina', 'sao paulo'),
    ('argentina', 'germany', 'rio de janeiro')]

```

```

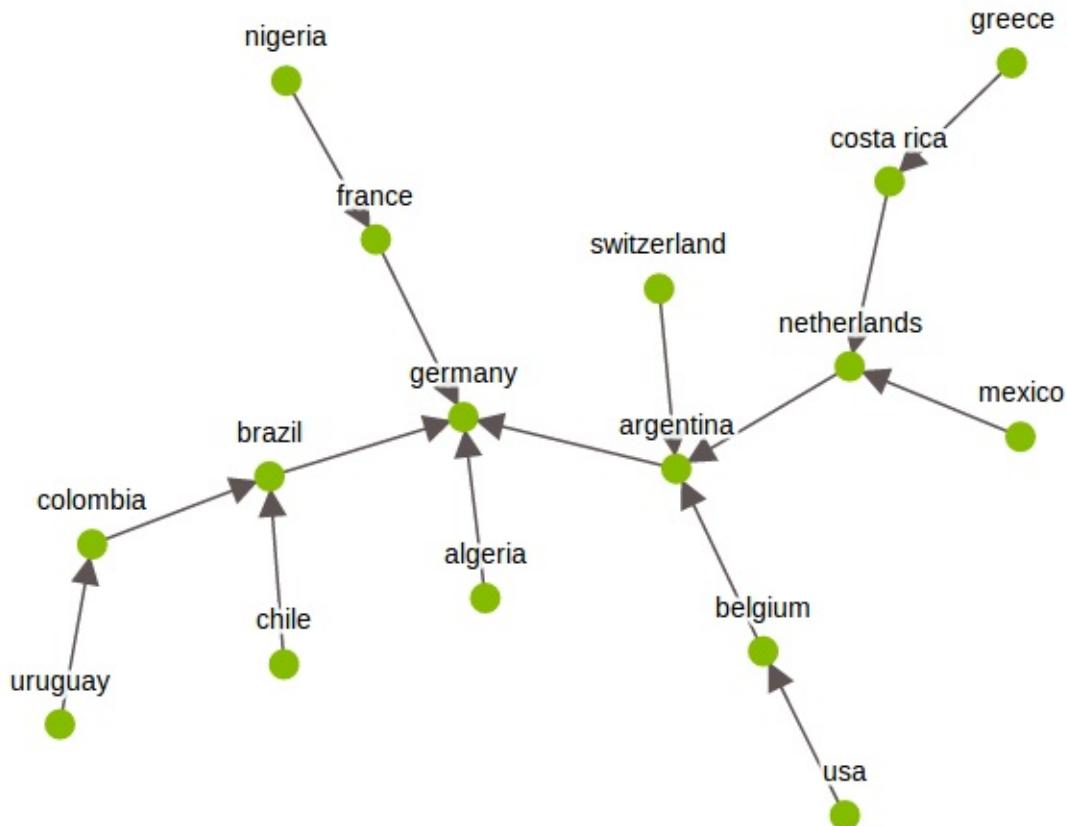
verts = graphlab.SFrame({'name': countries.keys(),
                        'continent': countries.values()})

losers, winners, locations = zip(*results)
edges = graphlab.SFrame({'loser': list(losers),
                        'winner': list(winners),
                        'location': list(locations)})

sg = graphlab.SGraph()
sg = sg.add_vertices(verts, vid_field='name')
sg = sg.add_edges(edges, src_field='loser', dst_field='winner')
print sg.summary()
sg.show(vlabel='id', arrows=True, ewidth=1.5)

```

```
{'num_edges': 15, 'num_vertices': 16}
```



A [connected component](#) of a graph is a subgraph where every pair of nodes is connected by some path through the component. For two nodes in *different* connected components there is no path between them.

The component assignment for each vertex is returned in both the 'componentid' and 'graph' objects. The former is an SFrame, while the 'graph' object is an SGraph whose vertices have an attribute with the answer. The following are equivalent ways to get the answer.

Question 2: How many connected components are there?

Because all countries are connected via the knockout stage results, there is only one giant connected component.

```
cc = graphlab.connected_components.create(sg)
print "number of components:", cc['component_size'].num_rows()
```

```
number of components: 1
```

Exercises

Predict if new questions asked on Stack Overflow will be closed

We all use [Stack Overflow](#), day in and day out, to get answers to our programming questions. The website, its content, and its users provide high quality solutions to reach other programmers. Quality is everything to them and they take it pretty seriously. In this exercise, we will build classifier to detect if questions will end up closed or not.

Closing Questions

Currently about 6% of all new questions end up **closed**. Questions can be closed as:

1. Off topic
2. Not constructive
3. Not a real question
4. Too localized

More in depth descriptions of each reason can be found in the [Stack Overflow FAQ](#). Your goal is to build a classifier that predicts whether or not a question will be closed using a lot of information about the post.

(source [Kaggle competition](#))

Data loading

Let us start by loading an SFrame given in binary format. Assuming the file given to you is in the location **FILEPATH**, then you can use the following:

```
sf = gl.SFrame(FILEPATH)
```

Canvas runs in the background, so you can continue to work as the summary visualizations are computed for you. In the mean time, let us take a quick look at the target column **OpenStatus**.

Question 1: Now, use the [SArray.unique\(\)](#) function to get out the unique values of the column **OpenStatus**.

```
unique_stats = sf['OpenStatus'].unique()
unique_stats.show()
```

It seems like the data was not very clean. That is mostly the case with real world datasets. We need to do some munging here.

Question 2: Add a column to the Sframe called **is_closed** which is 1 when **OpenStatus** is "closed" and 0 otherwise. *Hint:* You can use SArray's [boolean operation](#) or the [SArray.apply\(\)](#) function.

```
sf['is_closed'] = sf['OpenStatus'] == "closed"
```

Creating a balanced dataset.

It looks like only around **2.44%** of the data contains closed data. Classification on imbalanced data is a challenging task. Let us make a balanced dataset with an equal number of posts that are open and closed. One way of doing this is to **sub sample the data** with open posts to create more of a balance. We will accomplish this over the next few questions.

Question 3: Create an SFrame (lets call it **sf_closed_only**) that only contains the data in the original SFrame (**sf**) where **is_closed == 1**. Use the [SFrame's logical filter](#) to do so.

```
sf_closed_only = sf[sf['is_closed'] == 1]
```

Question 4: Let us use the [SFrame's sample](#) function to sample about 2.5% of the SFrame (**sf**) where **is_closed = 0**. Call the resulting SFrame **sf_open_only**.

```
sf_open_only = sf[sf['is_closed'] == 0].sample(0.025)
```

Question 5: Now, let us use [SFrame's append\(\)](#) functionality to append the SFrame **sf_open_only** with **sf_closed_only**. Call the resulting SFrame **sf_subsampled**.

NOTE: In some versions of IPython Notebook the following command must be run twice - the first time yields an error.

```
sf_subsampled = sf_open_only.append(sf_closed_only)
```

Question 6: Use the [SArray.astype\(\)](#) function to make sure the following columns (in **sf_subsampled**) are of the right types. Here are the right types.

- ReputationAtPostCreation (**int**)
- OwnerUndeletedAnswerCountAtPostTime (**int**)

```
sf_subsampled['ReputationAtPostCreation'] = sf_subsampled['ReputationAtPostCreation'].astype('int')
sf_subsampled['OwnerUndeletedAnswerCountAtPostTime'] = sf_subsampled['OwnerUndeletedAnswerCountAtPostTime'].astype('int')
```

Logistic Regression

In this section, we will create a classifier that can predict the target **is_closed** using the other features.

Question 7: Create a **train-test** split with 80% of the data being in the training set and 20% of the data in the test set. *Hint:* Using the [SFrame.random_split\(\)](#) function.

```
train_sf, test_sf = sf_subsampled.random_split(0.8)
```

The random split function creates two SFrames. One which we will use for training and a separate SFrame, which we will hold out for testing. The pattern of splitting your data for training and testing is important to ensure that your model is not too specialized for the training data.

Question 8: Use the training data and build a [logistic regression classifier](#) with the following features:

- *ReputationAtPostCreation*
- *OwnerUndeletedAnswerCountAtPostTime*

and the target **is_closed**. Use the default options for the logistic regression classifier.

```
model = gl.logistic_regression.create(train_sf, target='is_closed',
                                      features = ['ReputationAtPostCreation',
                                                   'OwnerUndeletedAnswerCountAtPostTime'])
```

Question 9: Use the [LogisticClassifier.evaluate\(\)](#) function to get useful statistics about the prediction on the test data (**test_sf**).

```
print model.evaluate(test_sf)
```

```
{'accuracy': 0.5653733528550512,
 'confusion_table': {'false_negative': 0,
                     'false_positive': 17811,
                     'true_negative': 0,
                     'true_positive': 23169}}
```

That was an accuracy of 56.52%. This isn't great. On closer inspection, you can see that the number of **false_positives** is too high. This means that the model is predicting all 1's. Let us try and see if we can get more out of our data.

Question 10: Collapse the columns `['Tag1', 'Tag2', 'Tag3', 'Tag4', 'Tag5']` into one column called `'tags_category'` (for convinience) as a **list type**. Use the `SFrame.pack_columns()` to collapse these columns.

Make sure you do this on **both the training and test data**.

```
train_sf = train_sf.pack_columns(column_prefix='Tag', dtype=list, new_column_name='tags_c
test_sf = test_sf.pack_columns(column_prefix='Tag', dtype=list, new_column_name='tags_cat
```

Question 11: Dictionaries are easier to work with, so let us convert the column `tags_category` to type dictionary using the `SFrame's apply()` function. The keys in the dictionary are the same as the elements in the list. The values of the dictionary are all set to 1.

For example, the list `["a", "b"]` must be converted to `{"a": 1, "b": 1}`

```
train_sf['tags_dict'] = train_sf['tags_category'].apply(lambda x: {a:1 for a in x})
test_sf['tags_dict'] = test_sf['tags_category'].apply(lambda x: {a:1 for a in x})
```

Feature Engineering

Let us try the `basic benchmark` from the Kaggle competition. Use the `SArray's apply` function to do the following things:

Question 12:

1. Create a column called **num_tags** that counts the number of keys in the columns `'tags_dict'`.
2. Create a column called **BodyMarkdown-Length** to count the length of the text in the column `'BodyMarkdown'`.
3. Create a column called **Title-Length** to count the length of the text of the column `'Title'`.

Note: Remember to do this on your train and test set.

```
# On the train data
train_sf['num_tags'] = train_sf['tags_dict'].apply(lambda x: len(x))
train_sf['BodyMarkdown-Length'] = train_sf['BodyMarkdown'].apply(lambda x: len(x))
train_sf['Title-Length'] = train_sf['Title'].apply(lambda x: len(x))

# On the test data
test_sf['num_tags'] = test_sf['tags_dict'].apply(lambda x: len(x))
test_sf['BodyMarkdown-Length'] = test_sf['BodyMarkdown'].apply(lambda x: len(x))
test_sf['Title-Length'] = test_sf['Title'].apply(lambda x: len(x))
```

Question 13:

Create a logistic regression model using the following features (on the data **train_sf**):

- ReputationAtPostCreation
- OwnerUndeletedAnswerCountAtPostTime
- num_tags
- BodyMarkdown-Length
- Title-Length

with the target being **is_closed**.

```
model = gl.logistic_regression.create(train_sf, target ='is_closed',
                                      features = ['ReputationAtPostCreation',
                                                  'OwnerUndeletedAnswerCountAtPostTime',
                                                  'num_tags',
                                                  'BodyMarkdown-Length',
                                                  'Title-Length'])
```

Question 14:

Again, evaluate your model on the test set (**test_sf**).

```
print model.evaluate(test_sf)
```

```
{'accuracy': 0.6072474377745242,
'confusion_table': {'false_negative': 3856,
'false_positive': 12239,
'true_negative': 5572,
'true_positive': 19313}}}
```

That's a bit better but not too much better. The **num_tags** had the highest coefficients so I suppose the tag data is very useful.

Gradient Boosted Trees

Let's see if we can do something in the non-linear space. Graphlab Create has recently added a fast **gradient boosting** module to train non- linear classifiers. It is a powerful model that can work quite well if you have a **few dense** features. If gradient boosting doesn't get us anywhere, then we probably want to engineer more features.

Question 15: Create a gradient boosted tree model (on **train_sf**) with the same features as the above logistic regression module. You must use the **objective as classification** and use the **same features as above** i.e:

- ReputationAtPostCreation
- OwnerUndeletedAnswerCountAtPostTime
- num_tags
- BodyMarkdown-Length
- Title-Length

Again, set the target as **is_closed**.

```
model = gl.boosted_trees.create(train_sf, target_column ='is_closed',
                                objective='classification',
                                feature_columns = ['ReputationAtPostCreation',
                                                    'OwnerUndeletedAnswerCountAtPostTime',
                                                    'num_tags',
                                                    'BodyMarkdown-Length',
                                                    'Title-Length'])

print model.evaluate(test_sf)

{'accuracy': 0.6542947888374329,
 'confusion_table': {'false_negative': 4873,
 'false_positive': 9294,
 'true_negative': 8517,
 'true_positive': 18296}}
```

That is much better. But I think we can do more if we use our data well.

Advanced Features: Using tag data

This is now a clear indication that we need to use some more features. How about we use the **tag_dict** as a feature directly. Graphlab Create's logistic regression module makes it super easy to add dictionaries into your model. Think of dictionaries as a collection of columns.

Hint for Question 17: For example, the dictionary `{a: 1, b:1, c:2}` feature implies that the feature `a` has value 1, feature `b` has value 1 and so on. Any features that are not explicitly in the dictionary are assumed to be zero. This way, you can only encode those features that are non-zero. This is perfect for the `tag` information because different posts may have a different number of tags and it is quite annoying to worry about what the total number of tags are.

Question 17: Now, create a **logistic regression** module with `tags_dict` as the feature as well as the same features as above:

- `tags_dict (dictionary)`
- `ReputationAtPostCreation`
- `OwnerUndeletedAnswerCountAtPostTime`
- `num_tags`
- `BodyMarkdown-Length`
- `Title-Length`

Note: Gradient boosted trees are not suitable for datasets with lots of features. Logistic regression is a better choice here.

```
model = gl.logistic_regression.create(train_sf, target ='is_closed',
                                      features = ['ReputationAtPostCreation',
                                                  'OwnerUndeletedAnswerCountAtPostTime',
                                                  'num_tags',
                                                  'tags_dict',
                                                  'BodyMarkdown-Length',
                                                  'Title-Length'])
```

Question 17: Again, evaluate the model on the test set (`test_sf`)

```
print model.evaluate(test_sf)
```

```
{'accuracy': 0.6728892142508541,
 'confusion_table': {'false_negative': 6209,
                     'false_positive': 7196,
                     'true_negative': 10615,
                     'true_positive': 16960}}
```

Bonus section: Text data

`Title` and `Body-Markdown` are two useful columns with raw text data. Let use the `count words` function to get some raw word counts.

Bonus question 1: Add a column called **title_word_count** that counts the number of words in the column **Title** and a column called **body_markdown_word_count** that adds the same for the column **BodyMarkdown**.

Note: Add this to both your test and train data.

```
# Train data
train_sf['title_word_count'] = train_sf['Title'].count_words()
train_sf['body_markdown_count'] = train_sf['BodyMarkdown'].count_words()

# Test data
test_sf['title_word_count'] = test_sf['Title'].count_words()
test_sf['body_markdown_count'] = test_sf['BodyMarkdown'].count_words()
```

Bonus question 2: Add the features **title_word_count** and **body_markdown_word_count** to the logistic regression classifier.

```
model = gl.logistic_regression.create(train_sf, target ='is_closed',
                                      features = ['ReputationAtPostCreation',
                                                  'OwnerUndeletedAnswerCountAtPostTime',
                                                  'num_tags',
                                                  'title_word_count',
                                                  'body_markdown_count',
                                                  'tags_dict',
                                                  'BodyMarkdown-Length',
                                                  'Title-Length'])
```

Bonus Question 3: Again, evaluate the model on the test set (**test_sf**)

```
print model.evaluate(test_sf)
```

```
{'accuracy': 0.704123962908736,
'confusion_table': {'false_negative': 6399,
'false_positive': 5726,
'true_negative': 12085,
'true_positive': 16770}}}
```

Exercises

The data for these exercises is culled from [Wikipedia's Database Download](#). Wikipedia's text and many of its images are co-licensed under the [Creative Commons Attribution-Sharealike 3.0 Unported License \(CC-BY-SA\)](#).

Load the first Wikipedia text file called "w0". Each line in the document represents a single document and there is no header line. Name the variable `documents`.

```
# Downloads the data from S3 if you haven't already.
import os
if os.path.exists('wikipedia_w0'):
    documents = graphlab.SFrame('wikipedia_w0')
else:
    documents = graphlab.SFrame.read_csv('https://static.turi.com/datasets/wikipedia/raw/')
    documents.save('wikipedia_w0')
```

Question 1:

Create an SArray that represents the documents in "bag-of-words format", where each element of the SArray is a dictionary with each unique word as a key and the number of occurrences is the value. *Hint:* look at the text analytics method `count_words`.

```
bow = graphlab.text_analytics.count_words(documents['X1'])
```

Question 2:

Create a trimmed version of this dataset that excludes all words in each document that occur just once.

```
docs = bow.dict_trim_by_values(2)
```

Question 3:

Remove all stopwords from the dataset. *Hint:* you'll find a predefined set of stopwords in `stopwords`.

```
docs = docs.dict_trim_by_keys(graphlab.text_analytics.stopwords(), exclude=True)
```

Question 4:

Remove all documents from `docs` and `documents` that now have fewer than 10 unique words. *Hint:* You can use SArray's logical filter.

```
ix = docs.apply(lambda x: len(x.keys()) >= 10)
docs = docs[ix]
```

Question 5: What proportion of documents have we removed from the dataset?

```
1 - ix.mean()
```

Topic Modeling

Question 6: Create a topic model using your processed version of the dataset, `docs`. Have the model learn 30 topics and let the algorithm run for 30 iterations. *Hint:* use the [topic modeling toolkit](#).

```
m = graphlab.topic_model.create(docs, num_topics=30, num_iterations=30)
```

Question 7: Print information about the model.

```
m
```

Question 8: Find out how many words the model has used while learning the topic model.

```
len(m['vocabulary'])
```

Use the following code to get the top 10 most probable words in each topic. Typically we hope that each list is a cohesive set of words, one that represents a general cluster of topics present in the dataset.

```
topics = m.get_topics(num_words=10).unstack(['word', 'score'], new_column_name='topic_word')
for topic in topics:
    print topic
```

Question 9: Predict the topic for the first 5 documents in `docs`.

```
m.predict(docs[:5])
```

Sometimes it is useful to manually fix words to be associated with a particular topic. For this we can use the `associations` argument.

Question 10: Create a new topic model that uses the following SFrame which will associate the words "law", "court", and "business" to topic 0. Use `verbose=False`, 30 topics, and let the algorithm run for 20 iterations.

```
fixed_associations = graphlab.SFrame()
fixed_associations['word'] = ['law', 'court', 'business']
fixed_associations['topic'] = 0
m2 = graphlab.topic_model.create(docs,
                                  associations=fixed_associations,
                                  num_topics=30, verbose=False, num_iterations=20)
```

Question 11: Get the top 20 most likely words for topic 0. Ideally, we will see the words "law", "court", and "business". What other words appear to be related to this topic?

```
m2.get_topics([0], num_words=20)
```

Transforming word counts

Remove all the documents from `docs` and `documents` that have 0 words.

```
tf_idf_docs = graphlab.text_analytics.tf_idf(docs)
```

Question 12: Use GraphLab Canvas to explore the distribution of TF-IDF scores given to the word "year".

```
tf_idf_docs.show()
```

Question 13: Create an SFrame with the following columns:

- `id` : a string column containing the range of numbers from 0 to the number of documents
- `word_score` : the SArray containing TF-IDF scores you created above
- `text` : the original text from each document

```
doc_data = graphlab.SFrame()
doc_data['id'] = graphlab.SArray(range(len(tf_idf_docs))).astype(str)
doc_data['word_score'] = tf_idf_docs
doc_data['text'] = docs
```

Question 14: Create a model that allows you to query the nearest neighbors to a given document. Use the `id` column above as your label for each document, and use the `word_score` column of TF-IDF scores as your features. *Hint:* use the new `nearest_neighbors`

toolkit.

```
nn = graphlab.nearest_neighbors.create(doc_data, label='id', feature='word_score')
```

Question 15: Find all the nearest documents for the first two documents in the data set.

```
nearest = nn.query(doc_data.head(2), label='id')
```

Question 16: Make an SFrame that contains the original text for the query points and the original text for each query's nearest neighbors. *Hint:* Use [SFrame.join](#).

```
nearest_docs = nearest[['query_label', 'reference_label']]
doc_data = doc_data[['id', 'text']]
nearest_docs.join(doc_data, on={'query_label':'id'})\
    .rename({'text':'query_text'})\
    .join(doc_data, on={'reference_label':'id'})\
    .rename({'text':'original_text'})\
    .sort('query_label')[['query_text', 'original_text']]
```

Exercises

In the code block below, import the StackOverflow dataset SFrame that you saved during earlier exercises. Note that [this data is shared](#) courtesy of StackExchange and is under the Creative Commons Attribution-ShareAlike 3.0 Unported License. This particular version of the data set was used in a recent [Kaggle competition](#).

```
import os
if os.path.exists('stack_overflow'):
    sf = graphlab.SFrame('stack_overflow')
else:
    sf= graphlab.SFrame('https://static.turi.com/datasets/stack_overflow')
    sf.save('stack_overflow')
```

Question 1: Visually explore the above data using GraphLab Canvas.

```
sf.show()
```

In this section we will make a model that can be used to recommend new tags to users.

Question 2: Create a new column called `Tags` where each element is a list of all the tags used for that question. (Hint: Check out `sf.pack_columns`.)

```
sf = sf.pack_columns(column_prefix='Tag', new_column_name='Tags')
```

Question 3: Make your SFrame only contain the `ownerUserId` column and the `Tags` column you created in the previous step.

```
sf = sf[['OwnerUserId', 'Tags']]
```

Question 4: Use the following Python function to modify the `Tags` column to not have any empty strings in the list.

```
def remove_empty(tags):
    return [tag for tag in tags if tag != '']
```

```
sf['Tags'] = sf['Tags'].apply(remove_empty)
```

Question 5: Create a new SFrame called `user_tag` that has a row for every (user, tag) pair.
 (Hint: See [sf.stack](#) .)

```
user_tag = sf.stack(column_name='Tags', new_column_name='Tag')
```

Question 6: Create a new SFrame called `user_tag_count` that has three columns:

- `OwnerUserId`
- `Tag`
- `Count`

where `Count` contains the number of times the given `Tag` was used by that particular `OwnerUserId` . Hint: See [groupby](#) .

```
user_tag_count = user_tag.groupby(['OwnerUserId', 'Tag'], graphlab.aggregate.COUNT)
```

Question 7: Visually explore this summarized version of your data set with GraphLab Canvas.

```
user_tag_count.show()
```

Creating a Model

Question 8: Use `graphlab.recommender.create()` to create a model that can be used to recommend tags to each user.

```
m = graphlab.recommender.create(user_tag_count, user_id='OwnerUserId', item_id='Tag')
```

Question 9: Print a summary of the model by simply entering the name of the object.

```
m
```

Question 10: Get all unique users from the first 10000 observations and save them as a variable called `users` .

```
users = user_tag_count.head(10000)['OwnerUserId'].unique()
```

Question 11: Get 20 recommendations for each user in your list of users. Save these as a new SFrame called `recs` .

```
recs = m.recommend(users, k=20)
```

When people use recommendation systems for online commerce, it's often useful to be able to recommending products from a single category of items, e.g. recommending shoes to somebody who typically buys shirts.

To illustrate how this can be done with GraphLab Create, suppose we have a Javascript user who is trying to learn Python. Below we will take just the Javascript users and see what Python tags to recommend them.

Question 12: Create a variable called `javascript_users` that contains all unique users who have used the `javascript` tag.

```
javascript_users = user_tag_count['OwnerUserId'][user_tag_count['Tag'] == 'javascript'].u
```

Question 13: Use the model you created above to find the 20 most similar items to the tag "python". Create a variable called `python_items` containing just these similar items.

```
python_items = m.get_similar_items(['python'], k=20)
python_items = python_items['similar']
```

Question 14: For each user in `javascript_users`, make 5 recommendations among the items in `python_items`.

```
python_recs = m.recommend(users=javascript_users, items=python_items, k=5)
```

Question 15: Use GraphLab Canvas to find out the 10 most often recommended items.

```
python_recs.show() # Then click on the Summary tab and look at the histogram in the sec
```

Question 16: Save your model to a file.

```
m.save('my_model')
```

Experimenting with new models

Question 17: Create a train/test split of the `user_tag_count` data from the section above.

Hint: Use `random_split_by_user`.

```
train, test = graphlab.recommender.util.random_split_by_user(user_tag_count,
                                                               user_id='OwnerUserId',
                                                               item_id='Tag')
```

Question 18: Create a recommender model like you did above that only uses the training set.

```
m1 = graphlab.recommender.create(train, user_id='OwnerUserId', item_id='Tag')
```

Question 19: Create a matrix factorization model that is better at ranking by setting `unobserved_rating_regularization` argument to 1.

```
m2 = graphlab.ranking_factorization_recommender.create(train,
                                                               user_id='OwnerUserId',
                                                               item_id='Tag',
                                                               target='Count',
                                                               ranking_regularization=1)
```

Question 20: Retrieve the coefficients for each user that were learned by this algorithm.

```
m2['coefficients']['OwnerUserId']
```

Question 21: Compare the predictive performance of the two models. Given the ability to make 10 recommendations, which model predicted the highest proportion of items in the test set (on average)?

```
results = graphlab.recommender.util.compare_models(test, [m1, m2],
                                                   metric='precision_recall')
```

Q: Why can't GraphLab Create read from my HDFS installation?

A: This usually happens because GraphLab Create failed to find a suitable java installation. Your server log (the location of this log is printed when GraphLab Create starts) will let you know if this is the issue.

If using GraphLab Create ≥ 1.3 , you can simply make sure JAVA_HOME is set to your preferred java installation. If you prefer to use a different java just for GraphLab Create, you can set GRAPHLAB_JAVA_HOME, and it will be checked before JAVA_HOME. If you prefer to use a JVM other than the standard one that comes with java, you'll have to specify the directory that holds libjvm.so with the environment variable GRAPHLAB_LIBJVM_DIRECTORY.

If using GraphLab Create < 1.3 , you can quickly solve this by upgrading :). If you don't want to upgrade, then you must set LD_LIBRARY_PATH to the directory that holds libjvm.so.

If nothing above fixes your issue, the problem may be that the version of Java is unsupported. Our HDFS support requires Java 6+.

Style Guide

In order to keep the appearance of the User Guide consistent, we are recommending the following guidelines:

Headers

- Each page has a top-level header(`# Linear Regression`)
- Headers in pages are fourth-level headers (`#### Experimentation Policies`)
- Fifth-level headers should only be used if they are in a fourth-level section.
- Capitalize headers according to
<http://web.archive.org/web/20130117225252/http://writersblock.ca/tips/monthtip/tipmar98.htm>

Text

- No line breaks within a paragraph
- When referring to APIs inline:
 - Wrap the string in backticks: `graphlab.deploy.map_job.create`
 - Use the entire namespace chain upon first mention, then just the member: `create`
 - Don't use parentheses, unless you include the entire signature.
 - Link an inline API to the public API docs on turi.com, at least upon first mention.

Code Snippets

- Annotate Python code with `python`
- For readability, include an empty line before the code snippet
- line breaks within code snippets: 80 columns