STUDENT'S GUIDE TO APACHE SPARK

Abhishek Devarakonda, Xiurong Lin, Ryan Borowicz, Jayanti Trivedi
MSBA6330 – Gold Cohort - Section 002
Team 5
December 14, 2016

Contents

1.0 Introduction

As aspiring data scientists in the University of Minnesota's MSBA program, we are being introduced to hundreds of different topics covering a wide spectrum from traditional statistics to machine learning to various coding languages. Our instructors do a great job of consolidating this information into a learnable format, but we will ultimately end up exploring external resources when researching further on various topics. The web is inundated with material on different subjects, and finding reputable sources and working coding examples can often be a challenge. We intend for this guide to be an introduction to Apache Spark for users of different experience levels, with a specific focus on the Spark SQL, ML, and SparkR modules. For beginners, there is enough overview material and applicable use cases to gain a high-level understanding, while for the more experienced users we have provided working code examples and tips that can be used to directly implement in your personal work. This tutorial incorporates the best content we have learned through our coursework and researching of external resources.

2.0 Tutorial Preparation Steps

2.1. Setting up a Working Spark Installation

Overview

Hortonworks provides a virtual machine (VM) with a pre-built Spark installation. For this tutorial, we will utilize Hortonworks Data Platform (HDP) 2.4, as the latest version (2.5) is still in Beta version and not as fully vetted with product integration. Additional methods to work through this tutorial include downloading a VM from a different provider (i.e. Cloudera), downloading Spark on your local machine (refer to Additional Resources section at the end of this document for installation instructions), or using a cloud provider such as Amazon Web Services (AWS).

General Information

- Initial Login Credentials
    - User = root, Password = hadoop (change on first login)
- Use the following http links to access various tools online
    - Online version of VM shell - http://127.0.0.1:4200/
    - Ambari VM Manager - http://127.0.0.1:8080
        - User = maria_dev, password = maria_dev

- iPython Notebook - http://127.0.0.1:8889
- Hortonworks Sandbox - http://127.0.0.1:8888/

Common Commands

- Run this command when opening the VM to enable python 2.7, which some packages require to run
  - source /opt/rh/python27/enable
- To access shell from ipython notebook
  - %%sh
  - ipython root directory is /usr/hdp/2.4.0.0-169/spark
- To send data from local desktop to VM – example using gitbash
  - scp -P <input-port> </input-directory-path-local-mach> <input-username@hostname-:/sandbox-dir-path>
  - scp -P 2222 ~/Downloads/iris.csv root@localhost:/root
- To send data from VM to local desktop – example using gitbash
  - scp -P <input-port> <input-username@hostname-:/sandbox-dir-path> </input-directory-path-local-mach>
  - scp -P 2222 root@localhost:/root/Collect.scala ~/Downloads
  - scp -P 2222 root@localhost:/usr/hdp/2.4.0.0-169/spark/twitter.ipynb ~/Downloads

Demo/Instructions

1) Download HDP 2.4 - https://hortonworks.com/downloads/
2) Load the ova file into your Oracle virtual box or VMware Player
3) Read this article and follow the provided instructions - https://hortonworks.com/wp-content/uploads/2016/02/Import_on_Vbox_3_1_2016.pdf
4) When initially logging in to the VM use the following credentials (user = root, password = hadoop)
5) Install ipython - http://hortonworks.com/hadoop-tutorial/using-ipython-notebook-with-apache-spark/
   a) yum install nano centos-release-SCL zlib-devel \
      bzip2-devel openssl-devel ncurses-devel \
      sqlite-devel readline-devel tk-devel \
      gdbm-devel db4-devel libpcap-devel xz-devel \
      libpng-devel libjpg-devel atlas-devel

b) yum groupinstall "Development tools"
c) yum install python27
d) source /opt/rh/python27/enable
e) wget https://bootstrap.pypa.io/get-pip.py
f) python get-pip.py
g) pip install numpy scipy pandas \
scikit-learn tornado pyzmq \
pygments matplotlib jsonschema
h) pip install jinja2 –upgrade
i) pip install jupyter
j) ipython profile create pyspark
k) nano ~/start_ipython_notebook.sh
l) #!/bin/bash - needs to be typed into vm - don't copy/paste
source /opt/rh/python27/enable
IPYTHON_OPTS="notebook --port 8889 \
--notebook-dir='/usr/hdp/current/spark-client/' \
--ip='*' --no-browser" pyspark
m) chmod +x start_ipython_notebook.sh
n) set up port forwarding for port8889 - refer to website link for details
o) Run the script from the shell - ./start_ipython_notebook.sh
p) Open your browser and go to http://127.0.0.1:8889 # this should open ipython notebook

Additional Resources

- Hortonworks Forum for VM Issues - https://community.hortonworks.com/topics/tutorial-380.html
- Hortonworks Getting Started Tutorials - https://hortonworks.com/tutorials/
- Stack Overflow Forum for Hortonworks VM Issues - http://stackoverflow.com/questions/tagged/hortonworks-sandbox?page=2&sort=newest&pagesize=15

3.0 Spark Module Overview & Tutorial – Using Python & R

3.1. Spark Basics

Spark RDD's

Resilient Distributed Datasets (RDD) are immutable distributed collection of objects and a fundamental data structure of Spark. Each dataset in an RDD is divided into logical partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes, and can be created through loading a file, parallelizing, or transforming an existing RDD. Examples of the first two methods are provided below:

- parallelize - lines = sc.parallelize(["pandas", "i like pandas"])
- lines = sc.textFile("/path/to/README.md")

Spark provides special operations on RDDs containing key/value pairs. These RDDs are called pair RDDs. Pair RDDs are a useful building block in many programs, as they expose operations that allow you to act on each key in parallel or regroup data across the network. Key/value pairs are an important aspect of pair RDD's that facilitate parallel processing and allow for a number of aggregation functions, including reduceByKey(), join(), groupByKey, combineByKey, mapValues(), flatMapValues(), keys(), values(), sortByKey(), subtractByKey, and cogroup. The code below provides an example of how to create a pair RDD:

- pairs = lines.map(lambda x: (x.split(" ")[0], x))

Some of the common functions for RDDs include transformations and actions, with some of the more frequent transformations detailed in the table below:

Transformations	Code Example
Filter	logs.filter(lambda line:".jpg" in line)
Map	squared = nums.map(lambda x: x * x).collect()
FlatMap	lines = sc.parallelize(["hello world", "hi"]) words = lines.flatMap(lambda line: line.split(" "))

Transformations are constructed by altering a previous RDD, and are considered lazy as no output is directed until an action is called. Actions compute results based on an RDD, and either return the result to the driver program or save it to an external storage system, such as HDFS. Some common actions are listed in the table below:

Actions	Code Example
Take, Collect, Show, First	Rdd.collect() Rdd.show() Rdd.first() For log in logs.take(5): print log
Reduce	sum = rdd.reduce(lambda x, y: x + y)
Count	Rdd.count()

After an RDD is created or transformed it can be cached or persisted in memory so the calculations don't need to be re-run each time it is called in an action. The lineage graph provides a way to track back RDD dependencies from the final action to all of the transformations done previously. The last key item you'll notice is the use of passing functions in python through lambda expressions, as well as top-level and locally-defined functions. Most functions involve the use of lambda, and apply map-like operations across the RDD.

3.1.1. Spark SQL

Overview:

Spark SQL is a Spark module for structured data processing. Spark SQL provides Spark with more information about the structure of both the data and the computation being performed, and uses this extra information to perform extra optimizations. There are several ways to interact with Spark SQL including SQL and the Dataset API. When computing a result the same execution engine is used, independent of which API/language you are using to express the computation. This unification means that developers can easily switch back and forth between different APIs based on which provides the most natural way to express a given transformation. A DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. DataFrames can be constructed from a wide array of sources, such as: structured data files, tables in Hive, external databases, or existing RDDs. Datasets provide additional features from RDDs, but Python does not currently have the support to use the dataset API. DataFrames are similar to Datasets except that DataFrames come in a named columnar format. Spark SQL introduces a new data abstraction called the SchemaRDD, which provides support for structured and semi-structured data. There are five features that make Spark SQL very powerful when performing exploratory data analysis:

1) **Integration** − Spark SQL with integrated APIs in Python, Scala and Java to let us query structured data as a distributed dataset (RDD) in Spark. This tight integration makes it easy to run SQL queries alongside complex analytic algorithms.
2) **Unified data access**− Spark SQL unify data access, since it could load and query data from a variety of sources. Schema-RDDs provide a single interface for efficiently working with structured data, including Apache Hive tables, parquet files and JSON files.
3) **Compatibility** − Spark SQL is compatible with hive, it could run unmodified Hive queries on existing warehouses. Plus, Spark SQL reuses the Hive frontend and MetaStore, compatibility with existing Hive data, queries, and UDFs.
4) **Connectivity** − Connect through JDBC or ODBC. Spark SQL includes a server mode with industry standard JDBC and ODBC connectivity.
5) **Scalability** − interactive and long queries use the same engine. Spark SQL support mid-query fault tolerance, letting it scale to large jobs too.

Demo/Tutorial

The following table provides the meetup.com data structure used in this Spark SQL example:

Column	Type	Description
Seqid	int	Eventid
Num_member	int	Member of evet
Country	string	Event country
State	string	Event state
city	int	Event city
Zipcode	float	Event zipcode
Latitude	float	Event latitude
Longitude	string	Event longitude

Step1: Load the data and create an RDD

Drag the csv file to your virtual machine, and move it to the home directory. Then, type in pyspark on the virtual machine shell, and create a new ipython file. Run the following code to create and check the first row of output.

```
data_file = "file:/home/training/training_materials/data/event.csv"
rawDataRDD = sc.textFile(data_file).cache()
rawDataRDD.take(1)
```

Output:

[u'12321,4,us,MN,Mpls,55401,44.98,-93.27']

Tips: Before uploading the data, make sure the dataset has been fully cleaned and pre-processed, or perform these operations in Spark. If there are any errors, you won't see these until running the query due to the "lazy" nature of RDDs. Specifically, each column should be only one data type rather than mixed data types. Typically, when creating a data frame, the data type defaults to string, and the string data type only accepts letters as strings, and no other special characters such as quotation, exclamatory, comma, and digital.

Step 2: Create a Spark context reference

To create a basic instance, a SparkContext reference is needed. Here we use the global context object sc for this purpose:

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
```

Step3: Create data frame from RDD

Before we create a data frame from existing RDD, we let Spark SQL know the schema of this dataset.

```
from pyspark.sql import Row
csvRDD = rawDataRDD.map(lambda l: l.split(","))
rowRDD = csvRDD.map(lambda p: Row(
    seqid=int(p[0]),
    num_member=int(p[1]),
    country=p[2],
    state=p[3],
    city=p[4],
    zipcode=int(p[5]),
    latitude=float(p[6]),
    longitude=float(p[7])
    ))
```

Step4: Register a temporary table

Before running a query, let`s infer and register the schema to create a table, then use printSchema to print out the schema.

```
meetupDF = sqlContext.createDataFrame(rowRDD)
meetupDF.registerTempTable("meetup")
meetupDF.printSchema()
```

Output:

```
root
|-- city: string (nullable = true)
|-- country: string (nullable = true)
|-- latitude: double (nullable = true)
|-- longitude: double (nullable = true)
|-- num_member: long (nullable = true)
|-- seqid: long (nullable = true)
|-- state: string (nullable = true)
|-- zipcode: long (nullable = true)
```

Step5: Query example 1

For our first example, we test registered the data frame by running a simple query as follows:

```
citypDF = sqlContext.sql("""
    SELECT city, max(num_member) as max_group, count(*) as event_cnt
    from meetup
    group by city
    order by max_group DESC
""")
citypDF.show(10)
```

Output:

City	Max_group	Event_cnt
Mpls	51	1524
SanFrancisco	37	1
SaintPaul	19	73
Hopkins	16	7
Chanhassen	13	3
ForestLake	12	2
Champlin	12	6
Seattle	10	1
Andover	9	7
SanAngelo	7	1

Step6: Query example 2

Hennepin county usually has a lot great events. Let`s check how large the size of meetup events have been hold in that area. We first filter the zip code as 55401, and choose city name,

number of members, longitude, and latitude. Then we sort the number of members in descending fashion to return the top 10 locations by number of members:

```
meetupDF.filter(meetupDF.zipcode==55401) \
.select("city","num_member","longitude","latitude").sort(meetupDF.num_member.desc()).show(10)
```

Output:

City	Num_member	Longitude	latitude
Mpls	51	-93.27	44.98
Mpls	50	-93.27	44.98
Mpls	50	-93.27	44.98
Mpls	43	-93.27	44.98
Mpls	43	-93.27	44.98
Mpls	41	-93.27	44.98
Mpls	39	-93.27	44.98
Mpls	37	-93.27	44.98
Mpls	37	-93.27	44.98
Mpls	33	-93.27	44.98

Step7: Query example 3

For this example, we would like to know the Minneapolis locations with the largest number of members. To accomplish this, we filter the city as Minneapolis, and use zip code to represent different areas, and which maximum numbers of member in each area, then sort by group size, and return the first 10 largest group size and its zipcode.

```
meetupDF.select("city","zipcode","num_member") \
    .filter(meetupDF.city=='Mpls') \
    .groupBy("zipcode") \
    .max("num_member") \
    .show(10)
```

Output:

zipcode	MAX(num_member#750L)
55431	48
55435	29
55437	12

zipcode	MAX(num_member#750L)
55438	10
55439	9
55436	7
55440	7
55433	7
55432	4
55434	1

Additional Resources

- https://www.infoq.com/articles/apache-spark-sql
- https://spark.apache.org/docs/latest/sql-programming-guide.html
- https://databricks-training.s3.amazonaws.com/data-exploration-using-spark-sql.html
- https://www.analyticsvidhya.com/blog/2016/09/comprehensive-introduction-to-apache-spark-rdds-dataframes-using-pyspark/
- https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-sql.html
- https://www.analyticsvidhya.com/blog/2016/10/spark-dataframe-and-operations/
- http://www.slideshare.net/jeykottalam/spark-sqlamp-camp2014

3.1.2. Spark ML

Overview:

Spark comes with a library containing common machine learning (ML) functionality, called Spark ML, previously MLlib. Spark ML provides multiple types of machine learning algorithms, including classification, regression, clustering, and collaborative filtering, as well as supporting functionality such as model evaluation and data import. All of these methods are designed to scale out across a cluster. Recently SparkML switched its primary data source from RDDs to dataframes to take advantage of the dataframe functionality, which provides a more intuitive interface and improved processing. In its current state, SparkML only supports parallel algorithms that run well on clusters, so it provides a somewhat limited set that should be mainly used on large datasets. For smaller datasets, other tools such as Python's sci-kit-learn or R should be used.

Data Types

Data Type	Description	Code

Dense Vector	Dense – floating-point number arrays	from numpy import array
		from pyspark.mllib.linalg import Vectors
		# Creates a dense vector (1.0, 4.0, 9.0)
		densevector = array([1.0, 4.0, 9.0]) # Numpy array

Sparse Vector	Sparse – store only non-zero values and indices (use if <= 10% of values are nonzero)	# Creates a sparse vector (1.0, 0.0, 2.0, 0.0)
		#using dictionary
		sparsevector1 = Vectors.sparse(4, {0: 1.0, 2: 2.0})
		#using two lists of indices/values
		sparsevector2 = Vectors.sparse(4, [0, 2], [1.0, 2.0])

LabeledPoint	local vector, either dense or sparse, associated with a label/response	from pyspark.mllib.linalg import SparseVector from pyspark.mllib.regression import LabeledPoint pos = LabeledPoint(1.0, [1.0, 0.0, 3.0])
Local Matrix	integer-typed row and column indices and double-typed values, stored on a single machine	from pyspark.mllib.linalg import Matrix, Matrices dm2 = Matrices.dense(3, 2, [1, 2, 3, 4, 5, 6])
Distributed Matrix	long-typed row and column indices and double-typed values, stored distributively in one or more RDDs	
Row Matrix	row-oriented distributed matrix without meaningful row indices, backed by an RDD of its rows, where each row is a local vector	from pyspark.mllib.linalg.distributed import RowMatrix rows = sc.parallelize([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]) mat = RowMatrix(rows)

Note: Some of the online Spark documentation is currently incorrect using “ml” instead of “mllib” in these examples

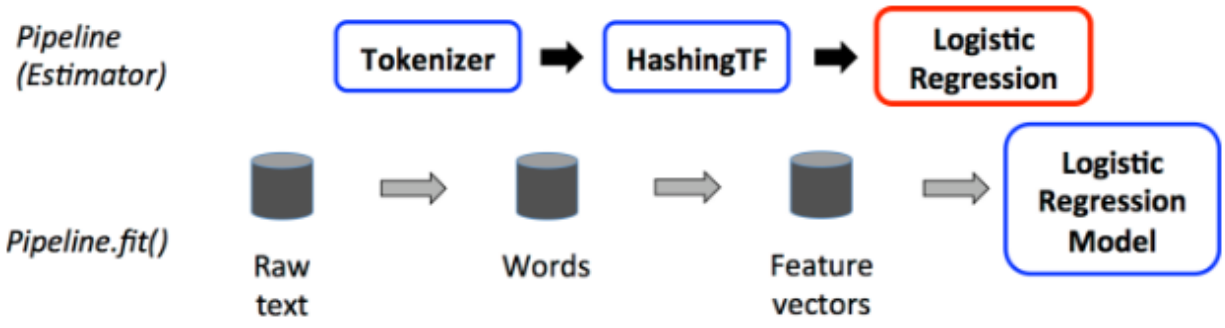
ML Pipelines

ML Pipelines provide the key integration framework integrating all of the different activities in machine learning, including pre-processing, model building, cross-validation techniques, and model evaluation. The terms used in Spark to address these items are as follows:

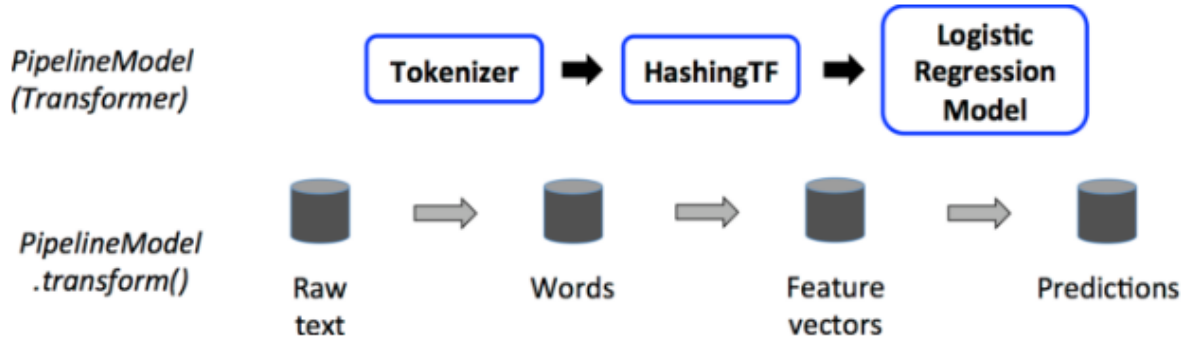
- **DataFrame** – uses SparkSQL DataFrame as dataset
  - Covered in section 3.1.1 in the Spark SQL section of this tutorial
- **Transfomer** – converts form one DataFrame to another
  - Ex. Adding new columns to dataframe, normalizing, etc.
- **Estimator** – algorithm fit on a DataFrame to produce a Transformer
- **Pipeline** – Chains transformers and estimators in an ML workflow
- **Parameter** – input values for the algorithms

The online Spark documentation provides a high-level diagram of the different steps of the pipeline:

Training Time:



Test Time:



SparkML provides a variety of different transformers that can be used to pre-process your data to prepare the data for use in the machine learning algorithms. Some of the key components are listed below:

Feature Extractors – Text Processing (TF-IDF, Word2Vec, CountVectorizer)

Feature Transformers:

- Dimension Reduction (PCA, DCT)
- Normalization/Standardization (Normalizer, StandardScaler, MinMaxScaler, MaxAbsScaler)
- Binning (Binarizer, Bucketizer, QuantileDiscretizer
- Text Processing (Tokenizer, StopWordsRemoer, n-gram)

- Other Userful Functions (VectorIndexer/VectorSlicer – column/row indexing, SQLTransformer – create new columns using sql functions, VectorAssembler– combine multiple columns into one)

Pre-Processing Examples:

Task	Description	Code

Descriptive Statistics	Summary dataset statistics	from pyspark.mllib.stat import Statistics
		Statistics.colStats(data) #statistical summary
		Statistics.corr(data1, data2, method) #correlation
		Statistics.chiSqTest(data)
		Statistics.mean(), .stdev(), .sum(), .sample(), .sampleByKey()

Normalizing	Spark provides different scaling functions to normalize your data based on min/max or standard deviation	from pyspark.mllib.feature import StandardScaler scale = StandardScaler() model = scale.fit(data) output = model.transform(data)
Feature Selection	Transformers that select most important features based on a variety of criteria	Ex. ChiSqSelector
Grid Search	Parameter Optimization	

Algorithms:

SparkML provides a number of common algorithms used in machine learning, although it should be noted that these are intended to be only those that can be parallelized and are used for big datasets.

- **Linear Regression** - Parameters (numIterations, stepSize, intercept, regParam)
- **Classification**
  - Uses LabelPoint objects
    - For Binary – Needs 0 & 1 as positive/negative labels,
    - 0 to number of classes – 1 for multi-class
  - Decision Trees – Parameters (data, numClasses, impurity, maxDepth, maxBins, categoricalFeaturesInfo)
  - Random Forest – Parameters (numTrees, fetaureSubsetStrategy, seed)
  - Logistic Regression – similar parameters to linear regression
    - Can use setThreshold() to change the threshold from 0.5 to address class imbalance problems
  - Support Vector Machine – Similar parameters to logistic regression

    - Naïve Bayes – uses multinomial naïve bayes
        - input features must not be negative
        - one parameter – lambda_
- **Unsupervised Learning**
    - K-Means Clustering – parameters (initializationMode, maxIterations, runs)
- **Recommenders** – use Rating objects
    - Collaborative Filtering - ALS – parameters (rank, iterations, lambda, alpha, numUserBlocks, numProductBlocks)
- **Dimensionality Reduction** - Currently not available in Python
    - Principal Component Analysis (PCA)
    - Singular Value Decomposition (SVD)
- **Text Processing** - TF-IDF
    - Hashing TF - run either on one document at a time or on a whole RDD. It requires each "document" to be represented as an iterable sequence of objects—for instance, a list in Python
    - IDF - You first call fit()on an IDF object to obtain an IDFModel representing the inverse document frequencies in the corpus, then call transform() on the model to transform TF vectors into IDF vectors
        - "from pyspark.mllib.feature import HashingTF

```
# Read a set of text files as TF vectors
    rdd = sc.wholeTextFiles("data").map(lambda (name, text): text.split())
    tf = HashingTF()
    tfVectors = tf.transform(rdd).cache()
    # Compute the IDF, then the TF-IDF vectors
    idf = IDF()
    idfModel = idf.fit(tfVectors)
    tfIdfVectors = idfModel.transform(tfVectors)"
```

[1]

Demo/Tutorial:

This demo will walk through setting up an end-to-end machine learning pipeline for a hospital patient dataset. This is a supervised learning problem that we will use logistic regression to try and predict the target value, which in this case is whether a patient will become infected with pneumonia or mrsa. The dataset includes 135 variables, and normally would be a good candidate to perform dimension reduction on. Unfortunately, SparkML does not currently

[1] (Karau, Konwinski, Wendell, & Zaharia, 2015)

support the dimension reduction algorithms in python, just scala. Below we have the detailed the steps to create the pipeline.

Step 1: Load the Data

```
training_file = "file:/usr/hdp/2.4.0.0-169/spark/final_patientyr_data_training_4.csv"
rawDataRDD = sc.textFile(training_file).cache()
```

Step 2: Create Row RDD and Convert to DataFrame

```
from pyspark.sql import SQLContext
from pyspark.sql import Row
sqlContext = SQLContext(sc)
csvRDD = rawDataRDD.map(lambda l: l.split(","))
rowRDD = csvRDD.map(lambda p: Row(
label=float(p[0]),
RXClass_Alt=int(p[1]),
RXClass_Antiinf=int(p[2]),
RXClass_Antineo=int(p[3]),
RXClass_Bio=int(p[4]),
RXClass_Cardio=int(p[5]),
RXClass_CNS=int(p[6]),
RXClass_Coag=int(p[7]),
RXClass_Gastro=int(p[8]),
RXClass_Horm=int(p[9]),
RXClass_Immo=int(p[10]),
RXClass_Metab=int(p[11]),
RXClass_Misc=int(p[12]),
RXClass_Nutri=int(p[13]),
RXClass_Psych=int(p[14]),
RXClass_Resp=int(p[15]),
RXClass_Top=int(p[16]),
RXClass_Unk=int(p[17]),
RXClass_UnkCat=int(p[18]),
PharmType_Drug=int(p[19]),
PharmType_Hosp=int(p[20]),
PharmType_Mail=int(p[21]),
PharmType_Online=int(p[22]),
PharmType_Store=int(p[23]),
PharmType_Unk=int(p[24]),
visit_count=int(p[25]),
VA=int(p[26]),
emer_stay=int(p[27]),
emer_med=int(p[28]),
oper=int(p[29]),
num_nights=int(p[30]),
emer_room=int(p[31]),
emer_rec=int(p[32]),
proc_count=int(p[33]),
```

```
Proc_11=int(p[34]),
Proc_13=int(p[35]),
Proc_14=int(p[36]),
Proc_16=int(p[37]),
Proc_18=int(p[38]),
Proc_2=int(p[39]),
Proc_20=int(p[40]),
Proc_21=int(p[41]),
Proc_22=int(p[42]),
Proc_23=int(p[43]),
Proc_24=int(p[44]),
Proc_27=int(p[45]),
Proc_28=int(p[46]),
Proc_3=int(p[47]),
Proc_31=int(p[48]),
Proc_33=int(p[49]),
Proc_35=int(p[50]),
Proc_36=int(p[51]),
Proc_37=int(p[52]),
Proc_38=int(p[53]),
Proc_39=int(p[54]),
Proc_4=int(p[55]),
Proc_40=int(p[56]),
Proc_42=int(p[57]),
Proc_44=int(p[58]),
Proc_45=int(p[59]),
Proc_46=int(p[60]),
Proc_47=int(p[61]),
Proc_48=int(p[62]),
Proc_49=int(p[63]),
Proc_50=int(p[64]),
Proc_51=int(p[65]),
Proc_53=int(p[66]),
Proc_54=int(p[67]),
Proc_55=int(p[68]),
Proc_57=int(p[69]),
Proc_58=int(p[70]),
Proc_59=int(p[71]),
Proc_6=int(p[72]),
Proc_60=int(p[73]),
Proc_62=int(p[74]),
Proc_63=int(p[75]),
Proc_64=int(p[76]),
Proc_65=int(p[77]),
Proc_66=int(p[78]),
Proc_67=int(p[79]),
Proc_68=int(p[80]),
Proc_69=int(p[81]),
Proc_70=int(p[82]),
Proc_74=int(p[83]),
Proc_75=int(p[84]),
```

```
Proc_76=int(p[85]),
Proc_77=int(p[86]),
Proc_78=int(p[87]),
Proc_8=int(p[88]),
Proc_80=int(p[89]),
Proc_81=int(p[90]),
Proc_82=int(p[91]),
Proc_83=int(p[92]),
Proc_84=int(p[93]),
Proc_85=int(p[94]),
Proc_86=int(p[95]),
Proc_87=int(p[96]),
Proc_88=int(p[97]),
Proc_89=int(p[98]),
Proc_92=int(p[99]),
Proc_93=int(p[100]),
Proc_94=int(p[101]),
Proc_96=int(p[102]),
Proc_97=int(p[103]),
Proc_98=int(p[104]),
Proc_99=int(p[105]),
Proc_ND=int(p[106]),
emer_rsn_Baby=int(p[107]),
emer_rsn_Babypre=int(p[108]),
emer_rsn_diag=int(p[109]),
emer_rsn_oth=int(p[110]),
emer_rsn_surg=int(p[111]),
emer_rsn_treat=int(p[112]),
cond_blood=int(p[113]),
cond_circ=int(p[114]),
cond_cong=int(p[115]),
cond_dig=int(p[116]),
cond_endo=int(p[117]),
cond_genit=int(p[118]),
cond_health_ctct=int(p[119]),
cond_ill_def=int(p[120]),
cond_infec=int(p[121]),
cond_poison=int(p[122]),
cond_mental=int(p[123]),
cond_musc=int(p[124]),
cond_NA=int(p[125]),
cond_neo=int(p[126]),
cond_nrvs=int(p[127]),
cond_NA2=int(p[128]),
cond_ND=int(p[129]),
cond_peri=int(p[130]),
cond_preg=int(p[131]),
cond_resp=int(p[132]),
cond_sense=int(p[133]),
cond_skin=int(p[134]),
cond_count=int(p[135])
```

```
    ))
training = sqlContext.createDataFrame(rowRDD)
training.registerTempTable("training")
#data_DF.printSchema()
```

Step 3: Use Vector Assembler to Create Features Vector

```
ignore = ['label']
assembler = VectorAssembler(
    inputCols=[x for x in training.columns if x not in ignore],
    outputCol='features')
assembler.transform(training)
```

Step 4: Create Logistic Regression Estimator

```
lr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)
```

Step 5: Create Pipeline

```
pipeline = Pipeline(stages=[assembler, lr])
```

Step 6: Create Parameter Grid

```
paramGrid = ParamGridBuilder() \
    .addGrid(lr.regParam, [0.1, 0.01]) \
    .build()
```

Step 7: Setup Cross-Validation

```
crossval = CrossValidator(estimator=pipeline,
                          estimatorParamMaps=paramGrid,
                          evaluator=BinaryClassificationEvaluator(),
                          numFolds=5)
```

Step 8: Run Cross-Validation, and Choose the Best Set of Parameters

```
cvModel = crossval.fit(training)
```

Step 9: Create Test File to Evaluate Predictions On

* same as Step 2, but remove the “label” variable and rename the datasets

Step 10: Make Predictions on the Test Data

```
prediction = cvModel.transform(test)
selected = prediction.select("probability", "prediction")
for row in selected.collect():
    print(row)
```

Additional Reference Materials:

- http://spark.apache.org/docs/latest/ml-guide.html
- Learning Spark (Karau, Konwinski, Wendell, & Zaharia, 2015)
- https://databricks-training.s3.amazonaws.com/movie-recommendation-with-mllib.html
- https://www.codementor.io/spark/tutorial/building-a-recommender-with-apache-spark-python-example-app-part1
- https://github.com/apache/spark/tree/master/examples/src/main/python/mllib
- http://stanford.edu/~rezab/sparkworkshop/slides/xiangrui.pdf

3.1.3. SparkR

Overview:

SparkR is an R package that provides a light-weight frontend to use Apache Spark from R. In Spark 2.0.2, SparkR provides a distributed data frame implementation that supports operations like selection, filtering, aggregation etc. (similar to R data frames, dplyr) but on large datasets. SparkR also supports distributed machine learning using MLlib. A SparkDataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R, but with richer optimizations under the hood. SparkDataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing local R data frames.

Demo/Tutorial:

- Refer to the project portfolio for the online notebook associated with this tutorial.

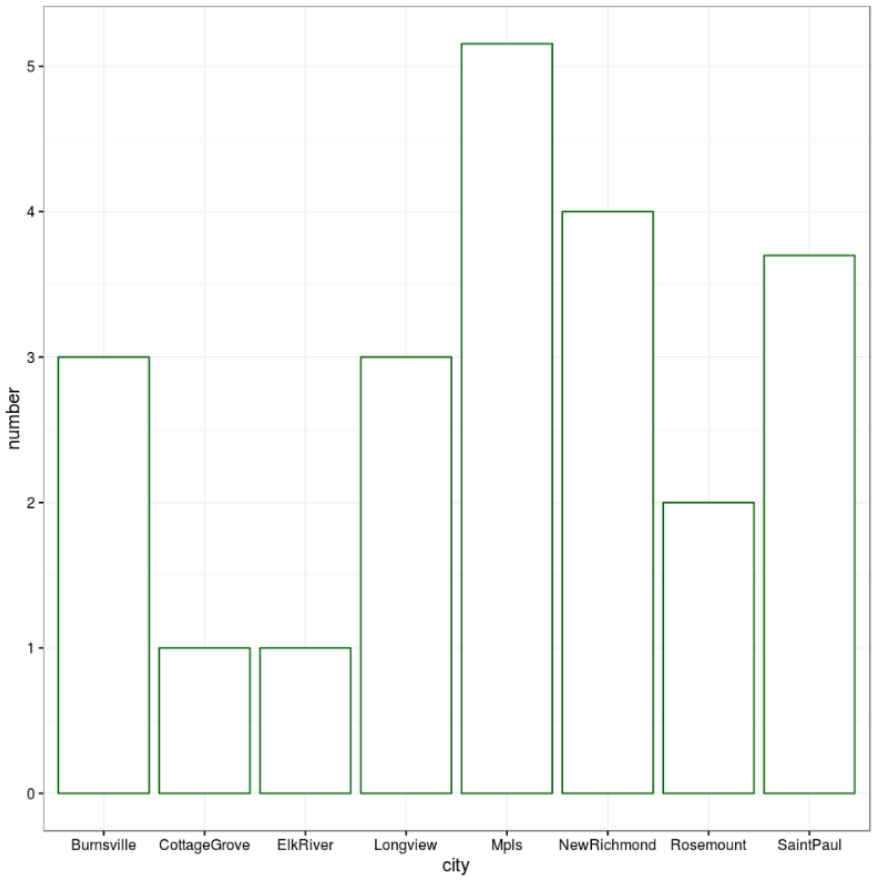
Step 1: Load Data

```
meetup_mpls<- read.df(sqlContext, source = "csv", path =
"/FileStore/tables/sz2a9k1c1481670199696/", delimiter = ",",header="true", inferSchema =
"true")
cache(meetup_mpls)
summary(meetup_mpls)
head(meetup_mpls)
```

	seqid	number	country	state	city	zipcode	long	lan
1	12321	4	us	MN	Mpls	55401	44.98	-93.27
2	12323	1	us	MN	Mpls	55401	44.98	-93.27
3	12391	1	us	MN	Mpls	55455	44.97	-93.24
4	12395	5	us	MN	Mpls	55401	44.98	-93.27
5	12397	4	us	MN	Mpls	55401	44.98	-93.27
6	12398	4	us	MN	Mpls	55401	44.98	-93.27

Step 2: Data Visualization

```
library(ggplot2)
citylist<-
c("Mpls","NewRichmond","SaintPaul","Rosemount","Longview","Burnsville","ElkRiver","CottageGrove")
meetup_mpls1 <- subset(meetup_mpls,meetup_mpls$city %in% citylist)
numberBycity <- meetup_mpls1 %>% group_by("city") %>% avg("number") %>%
withColumnRenamed("avg(number)", "number")
ggplot(collect(numberBycity), aes(city, number)) + geom_bar(fill="white", colour="darkgreen",stat =
"identity") + theme_bw()
```

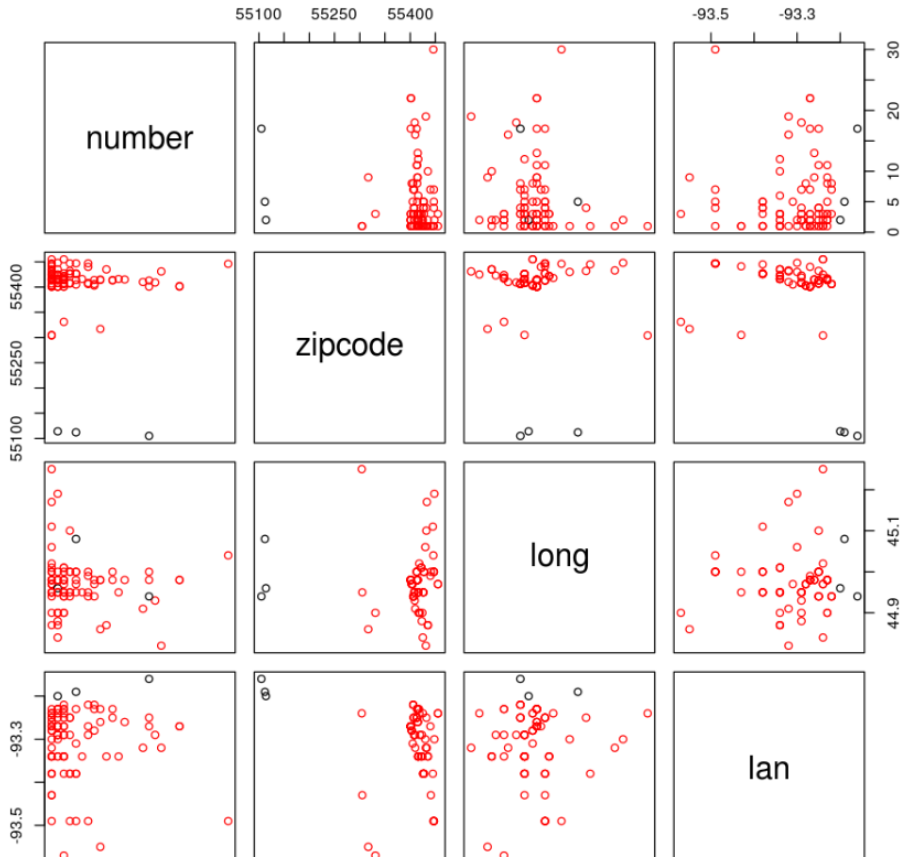


Step 3: Clustering

```
library(magrittr)
clusters <- spark.kmeans(meetup_mpls, ~ number + zipcode+long+lan, k = 3, initMode =
"random")
summary(clusters)
```

```
pred <- predict(clusters, meetup_mpls) %>% subset(select = c("number", "zipcode", "long",
"lan","prediction")) %>% sample(withReplacement = F, fraction = 0.05) %>% collect

pairs(pred[1:4], cex = 1.0, col = pred$prediction + 1)
```



Step 4: Regression

```
training <- subset(meetup_mpls, meetup_mpls$zipcode > 55401 )
testing <- subset(meetup_mpls, meetup_mpls$zipcode < 55401)
m <- glm(number ~ zipcode + long + lan + city,family = "gaussian",  data = training)
summary(m)

training.preds <- m %>% predict(training)
preds <- training.preds %>% subset(select = c("prediction", "number")) %>% collect
cor(preds$number, preds$prediction)^2

training <- subset(songs, songs$year < 2008 & songs$year > 0)
testing <- subset(songs, songs$year >= 2008)
model <- glm(loudness ~ tempo + year + end_of_fade_in + duration, family = "gaussian", data =
training)
summary(model)
```

```
training.preds %>%  subset(select = c("city", "zipcode", "long", "lan", "prediction"))  %>%
sample(F, 0.05) %>% collect %>% display
```

city	zipcode	long	lan	prediction
Mpls	55418	45.02	-93.24	4.688472376878622
Mpls	55406	44.94	-93.22	4.990014047550801
Mpls	55441	45	-93.43	4.4080790734181505
Mpls	55413	45	-93.25	4.7416740847691585
Mpls	55422	45.01	-93.34	4.5497369573215565
Mpls	55415	44.98	-93.26	4.785803995387596
Mpls	55415	44.98	-93.26	4.785803995387596
Mpls	55414	44.98	-93.23	4.83665127013694
Mpls	55416	44.95	-93.34	4.747229645353798

Additional Reference Materials:

- http://spark.apache.org/docs/latest/sparkr.html
- https://github.com/amplab-extras/SparkR-pkg
- http://amplab-extras.github.io/SparkR-pkg/

3.1.4. Spark Streaming

Overview:

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources like Kafka, Flume, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window. Spark Streaming provides a high-level abstraction called discretized stream or DStream, which represents a continuous stream of data. DStreams can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs. In this project, we initially looked into using spark streaming to pull real time data through the meetup.com API. While we were not able to complete the integration due to technical difficulties, we did accomplish pieces of the process.

API Overview:

Connectivity is an amazing thing. We all are now used to the instant connectivity that puts the world at our finger tips. We can purchase, post, pick anything and anywhere. We are connected to each other like never before. But how does data get from here to there? How do different devices connect to each other? All of this is due to what's called an Application Programming interface, or API.

API is what makes possible all the interactivity we come to expect and rely on. It is a messenger that takes requests and tells the system what you want to do and returns the response back to us. The best example is if we are booking a flight from an online travel booking website that aggregates information from various airlines. Here the travel website interacts with airline's API. The API is an interface that can be asked by the online service, to get the information from the airline system over the internet to book seats and other needs. It also then then takes the airline's response and delivers it right back to the online service that is then shown to us.

The same goes to any interaction between application, data, and devices. They all have APIs that allow computers to operate them and that's what creates connectivity. It is like a waiter's job in a hotel that connects customers to the kitchen to get the food ordered.

Based on how they are used, APIs are divided into 3 types:

- Internal (Private): These are APIs that are just for internal usage in a company. No one from outside can use this.
- Partner: These are the APIs that only specific companies or business partners can use. The give access to business requirements or functions, as per the partnership between the companies. For example, online travel booking websites, need access to the database of flights or hotels to check the availability of flights/hotels. In such cases, the companies use partner APIs
- External (Public): These APIs are available for public use. This is done to facilitate newer applications or capabilities to leverage the services.

Demo/Tutorial:

Spark streaming is an extension of the Spark API that specializes in handling live data streams. It enables high scalability, fault tolerance, and high throughout. It is compatible to work with other sources of data like Kafka, HDFS/ S3, Flume, Kinesis etc. To work on spark streaming, we need to use spark context, which is the main point of start for all streaming functionality.

```
from pyspark import SparkContext
```

```
from pyspark.streaming import StreamingContext

# Create a local StreamingContext with two working threads and batch interval of 1 second

sc = SparkContext("local[2]", "NetworkWordCount")

ssc = StreamingContext(sc, 1)
```

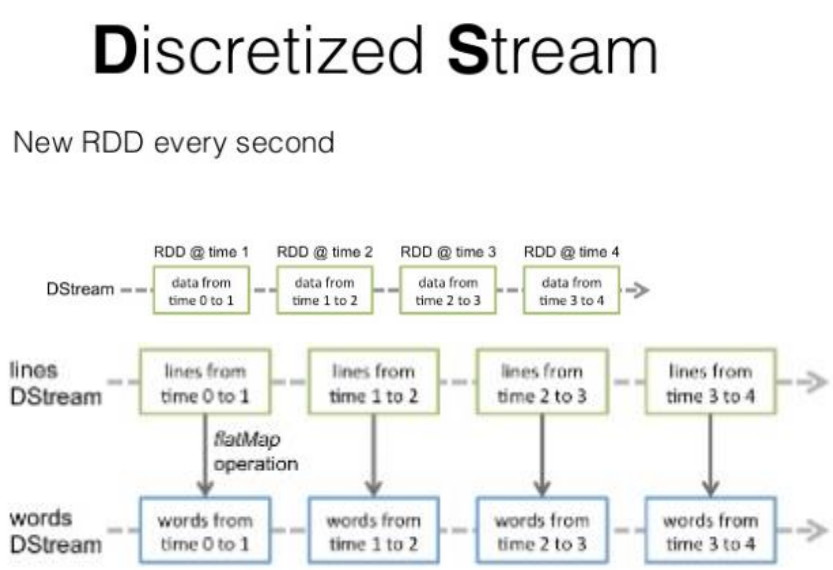
Then we create a DStream that represents a streaming data from a host / URL:

```
# Create a DStream that will connect to hostname:port, like localhost:9999

lines = ssc.socketTextStream("localhost", 9999)
```

To start taking the data and working the code on it, we need to start the streaming context and then terminate in some form. Else the code goes on and on.

- `ssc.start()` `# Start the computation`
- `ssc.awaitTermination()` `# Wait for the computation to terminate`



Additional Reference Materials:

- http://spark.apache.org/docs/latest/streaming-programming-guide.html
- https://github.com/apache/spark/tree/master/examples/src/main/scala/org/apache/spark/examples/streaming
- http://people.csail.mit.edu/matei/papers/2013/sosp_spark_streaming.pdf

- http://www.slideshare.net/SergeyZelvenskiy/spark-streaming-45954549
- https://stanford.edu/~rezab/sparkclass/slides/td_streaming.pdf

3.1.5. Plotly Visualization

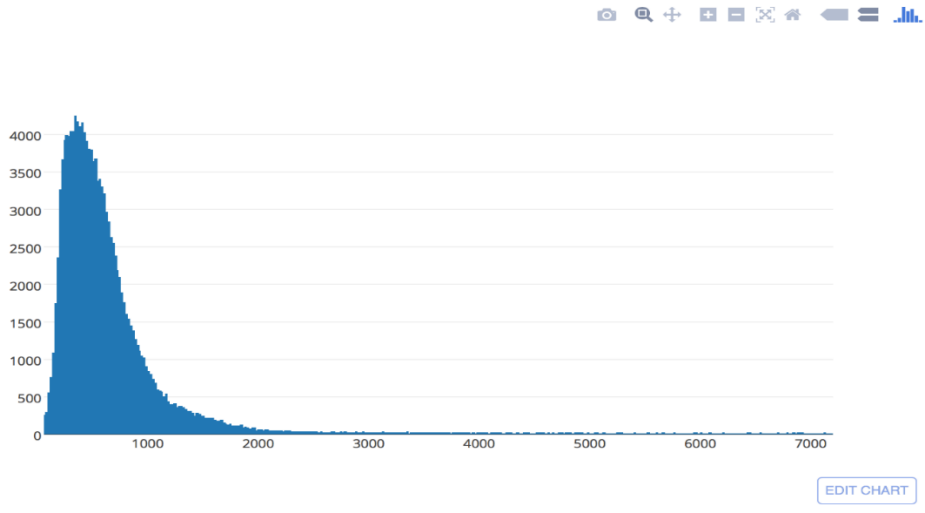
Plotly's ability to graph and share images from Spark DataFrames quickly and easily make it a great tool for analysis and visualizations. The functionalities of Plotly can be used by importing the below packages:

```
import plotly.plotly as py
from plotly.graph_objs import *
```

Using a sample dataset of open bike rental data, we will try to show we can make histograms in plotly. Below is the command used to make the histogram,

```
data = Data([Histogram(x=df2.toPandas()['d1'])])
py.iplot(data, filename="spark/less_2_hour_rides")
```

where df2 is the dataframe having open bike rental data and d1 is the column name which has numerical data. On running the above command, below graph is obtained, which can be modified in the ipython notebook in various ways. As can be seen, on the top right corner of the graph, it can be zoomed in and out, auto-scaled, axis can be rescaled, and data can be compared by hovering. The graph is an interactive image which shows numbers when hovering over it. Below is a screenshot of how the plotly histogram looks in the ipython notebook.

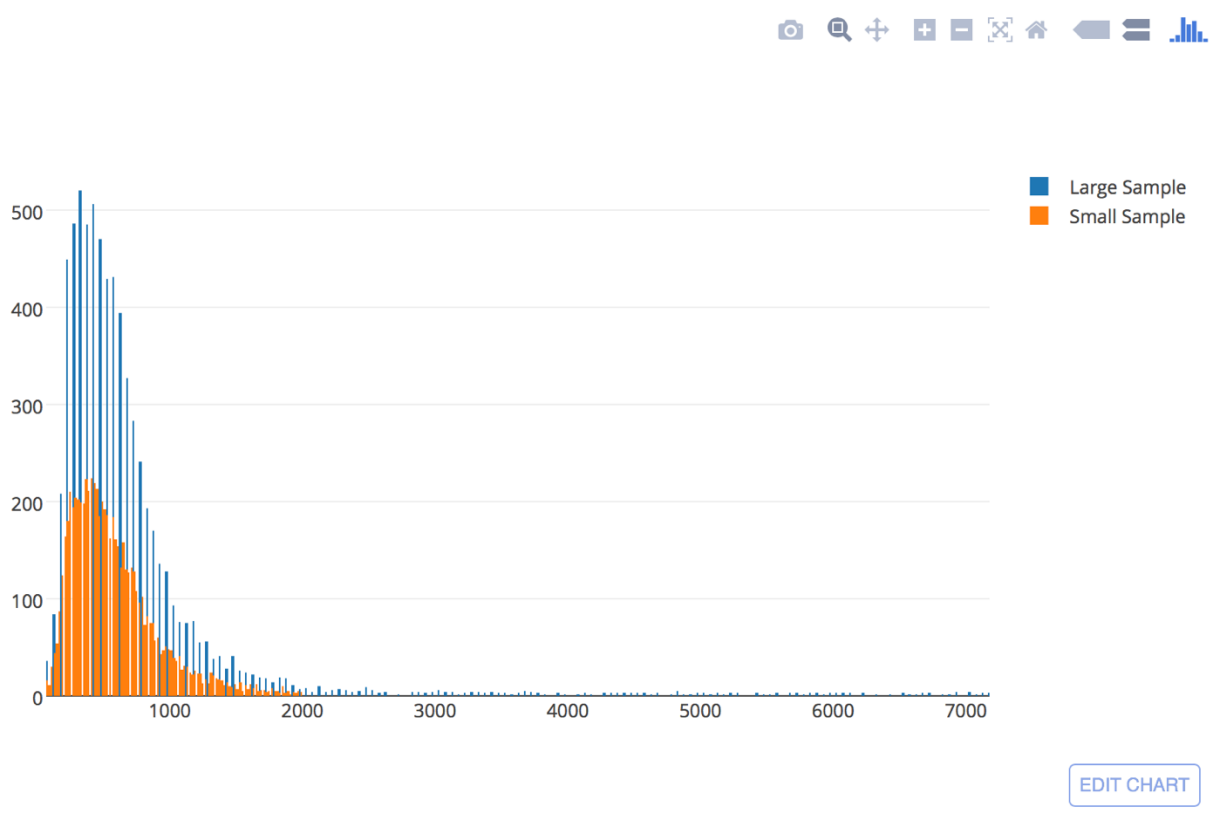


Overlayed histograms can also be made using plotly. This is a great way to eyeball different distributions. It gives us chance to study distribution of two different numerical variables and compare them. Below is the code to create two histograms on one frame:

```
data = Data([
    Histogram(x=s1.toPandas()['d1'], name="Large Sample"),
    Histogram(x=s2.toPandas()['d1'], name="Small Sample")
])

py.iplot(data, filename="spark/sample_rides")
```

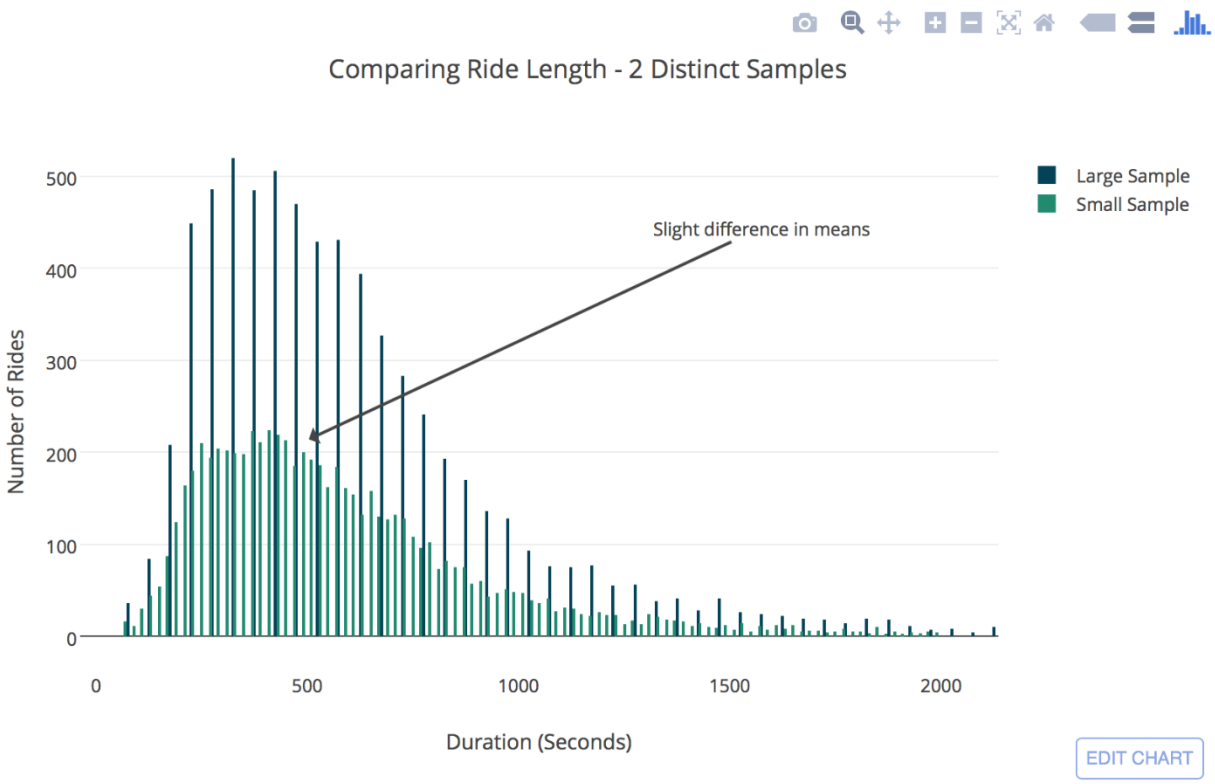
Below is the screenshot of overlayed histograms drawn from the above code: -



As below, it can be seen how two histograms are plotted together on one graph on two different samples of the same dataframe.

What's really powerful about Plotly is sharing this data is simple. We can take the above graph and change the styling or bins visually. A common workflow is to make a rough sketch of the graph in code, then make a more refined version with notes to share with management.

Another graph below is a more refined version which is more descriptive: -



Next, by converting a Spark Dataframe to pandas dataframe and then making a graph for top 3 groups. Y axis displays the calculated count. Below is the code:

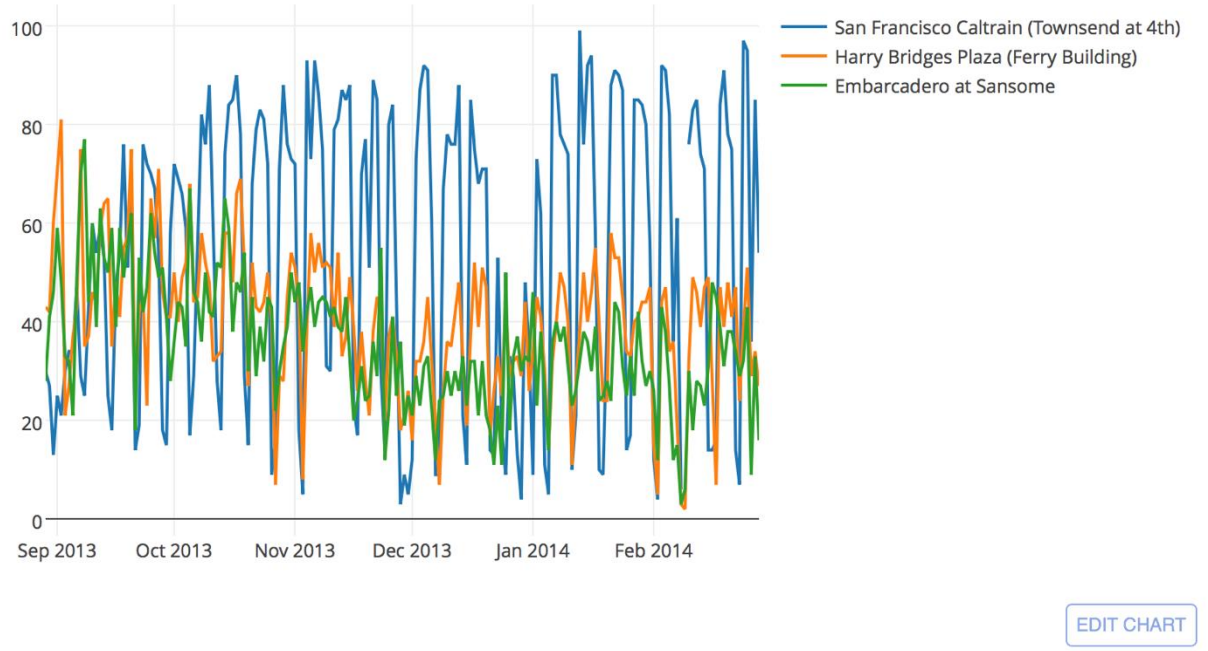
```
dep_stations = btd.groupBy(btd['Start Station']).count().toPandas().sort('count',
ascending=False)

def transform_df(df):
    df['counts'] = 1
    df['Start Date'] = df['Start Date'].apply(pd.to_datetime)
    return df.set_index('Start Date').resample('D', how='sum')
```

```
pop_stations = [] # being popular stations - we could easily extend this to more stations
for station in dep_stations['Start Station'][:3]:
    temp = transform_df(btd.where(btd['Start Station'] == station).select("Start Date").toPandas())
    pop_stations.append(
        Scatter(
        x=temp.index,
        y=temp.counts,
        name=station
        )
    )

data = Data(pop_stations)
py.iplot(data, filename="spark/over_time")
```

Below is the graph which shows count of all three groups overlayed on each other over a time period:



4.0 Concluding Remarks

Our team covered a lot of ground over the course of this project, and really extended our learning around Spark and its multi-faceted capabilities for big data processing. While still new to the scene, Spark provides enough functionality to complete an end-to-end data science project. It currently lags a little bit in the ease of pre-processing data as well as data visualization, but is improving on these capabilities through the use of SparkR and extensions like Plotly. As a product that many companies are using today and should only continue to increase its use going forward, this was a great learning experience for our team, and should serve to benefit our education and careers going forward. We hope the content, tutorials, and reference materials will provide a useful guide for our fellow students in exploring the different features of Apache Spark.

5.0 Bibliography

ampcamp. (2015). *Big Data Bootcamp*. Retrieved from AMPcamp: http://ampcamp.berkeley.edu/

Apache Spark. (2016). *Machine Learning Library (MLlib) Guide*. Retrieved from Apache Spark: http://spark.apache.org/docs/latest/ml-guide.html

Apache Spark. (2016). *Spark SQL, DataFrames and Datasets Guide*. Retrieved from Apache Spark: http://spark.apache.org/docs/latest/sql-programming-guide.html

Apache Spark. (2016). *Spark Streaming Programming Guide*. Retrieved from Apache Spark: http://spark.apache.org/docs/latest/streaming-programming-guide.html

Apache Spark. (2016). *SparkR (R on Spark)*. Retrieved from Apache Spark: http://spark.apache.org/docs/latest/sparkr.html

Databricks. (2014). *Databricks Reference Apps*. Retrieved from gitbook: https://www.gitbook.com/book/databricks/databricks-spark-reference-applications/details

Dianes, J. A. (2016, June). *spark-py-notebooks*. Retrieved from Github: https://github.com/jadianes/spark-py-notebooks

Karau, H., Konwinski, A., Wendell, P., & Zaharia, M. (2015). *Learning Spark*. O'Reilly Media.

Kutianski, J. P. (2016, July 9). *Using the Meetup API*. Retrieved from Github: https://github.com/jkutianski/meetup-api/wiki

Meetup. (2016). *Meetup API*. Retrieved from Meetup: http://www.meetup.com/meetup_api/

Meng, X. (n.d.). *MLlib: Scalable Machine Learning on Spark*. Retrieved from stanford.edu: http://stanford.edu/~rezab/sparkworkshop/slides/xiangrui.pdf

Penchikala, S. (2015, April 16). *Big Data Processing with Apache Spark - Part 2: Spark SQL*. Retrieved from infoq: https://www.infoq.com/articles/apache-spark-sql

Plotly. (2015). *Spark Dataframes with Plotly*. Retrieved from https://plot.ly/python/apache-spark/

R frontend for Spark. (2015). Retrieved from Github: https://github.com/amplab-extras/SparkR-pkg

Ray, S. (2015, August 19). *Analytics Vidhya*. Retrieved from Data scientist hack to find the right Meetup groups (using Python): https://www.analyticsvidhya.com/blog/2015/08/data-scientist-meetup-hack/

Tran, A. B. (2015, April 3). *Tutorial: Using Python and R to pull and analyze Meetup.com API data*. Retrieved from trendct: http://trendct.org/2015/04/03/using-python-and-r-to-pull-and-analyze-meetup-com-api-data/