DIC LAB 3 - CRIME PREDICTION & INSIGHT GENERATION

Malvika, and David Jegan
Department of Computer Science
University at Buffalo
{msundara,davidjeg}@buffalo.edu

***Abstract*— The primary focus of our project is to predict the category of the crime based on the geo-spatial and temporal dimensions of the event. Leveraging the publicly available San Francisco city crime dataset we can derive insights and predict the levels of democracy, hot-spots of violence and geographical spread of disturbance etc. The dataset is humongous in terms of GB, and spark helps to bring ML model to the data, increasing the performance, and scalability.**

I. Problem Statement

Crime rate is an ubiquitous problem in all metropolitan cities and San Francisco (SFO) is no exception. In spite of the scale of geographical extent of events, we can observe that certain type of crimes are prevalent within certain spatial locations. If we are able to predict the event type and form relationship between the individual events, we can gain great insights from the data. The overall dataset is huge and contains around 2.3 Million entries, thus Apache Spark is a powerful tool with many useful libraries (like MLlib and GraphX) and it would be a best fit solution for running an ML model on data of this size. The highly connected data can be constructed into a graph, to help us proactively prevent crime rather than being reactive after the incident has occurred.

II. Data Source

The crime data source is downloaded from the San Francisco government data (https://data.sfgov.org/Public-Safety/) which is publicly available on the internet. It is an aggregated collection of crime and miscreant information garnered by the SFO Police department released and updated periodically. This public crime data does not include any sort of information about persons related to crimes, not even as anonymity tokens - it supplies only crime and location data. We tried loading the data in Jupyter Notebook but it took around 5 minutes to just load the data, whereas in Spark cause of lazy evaluation, in-memory computation and parallel processing, loading data was much quicker ie. less than a minute.

A. *Size*

There are around 2.3 million records and the overall size is around 620MB.

B. *Duration*

The data is available from March 2013 to current day (Month of April 2019). The data is updated in a weekly fashion.

C. *REST API call*

The upcoming new events can be extracted using the API, the SFO government has provided. It can be downloaded in csv or json format. [4] They have deployed their API under Socrate Open Data (SODA) and we can use it for filtering, querying and aggregating data.

III. Solution Architecture

Fig. 1. Overview

A. *Environment, Tools and Technologies*

1) **Data collection** Jupyter Notebook with Python runtime
2) **Graph interpreter** Neo4J is chosen as the base database as the data is rich and connected
3) **Spark** PySpark is used as the runtime environment. PySparkSQL is used for data manipulation.
4) **Machine Learning** Spark MLLib is used for implementing Random forest, Naive Bayes classifier and Logistic Regression
5) **Graphx** Triangle Count and Betweenness Centrality algorithms are implemented using Spark Graphx.The scala API is used here. Betweeness Centrality is implemented using the spark-betweenness package.
6) **Visualization** Google Cloud Datastudio and Neo4J Graph visualization is used here

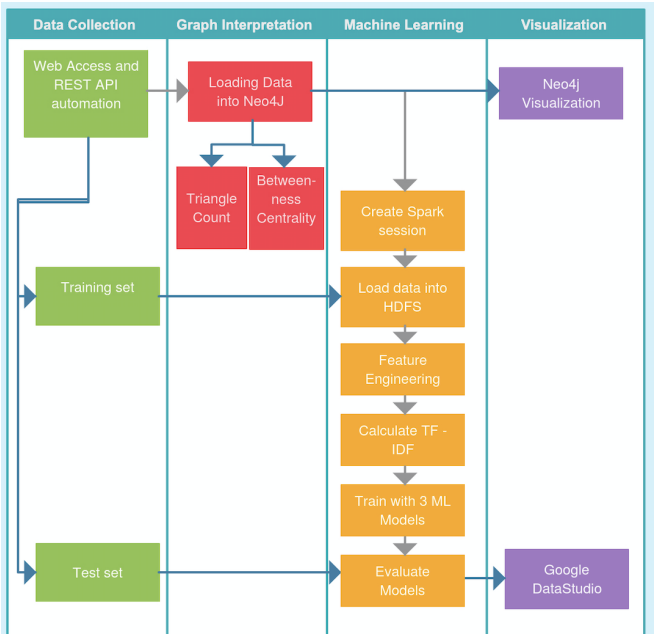


Fig. 2. Detailed Overall Flow

B. *Data Collection*

We have written a subroutine to automatically access the data source in a programmatic manner using API call. The figure 4 below demonstrates how to extract data using their API.

1) Data Model: We consider the following event model as our datasource is rich in interconnection as shown in the figure 3, where events is the entity to be predicted and objects, locations are the features which would be connected. As shown in the figure, the data is interconnected.

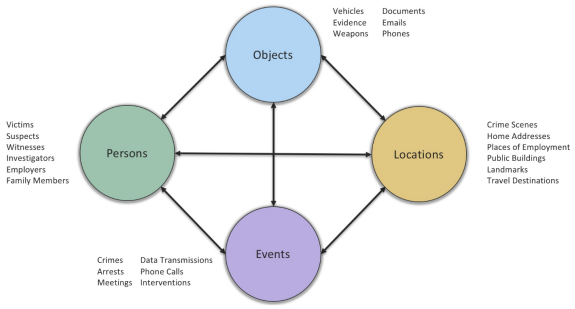
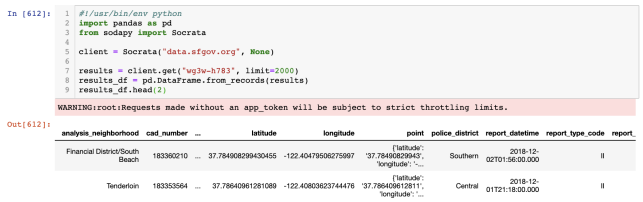


Fig. 3. Data Interconnection Sample

C. *Graph Interpretation*

The dataset here is humongous and utilizing spark's inherent ideology of moving code to data helps in improving the overall performance of the system. This data structure is more inclined for a graph DB as it is represents a graph. Therefore, we store the data to Neo4J Graph database and leveraging the Neo4J Spark connector we can load the data from neo4J DB directly into spark and store the results back into Neo4J as graphs, as shown in figure 5, 6 and 7 below.



```
In [612]:
#!/usr/bin/env python
import pandas as pd
from sodapy import Socrata

client = Socrata("data.sfgov.org", None)

results = client.get("wg3w-h783", limit=2000)
results_df = pd.DataFrame.from_records(results)
results_df.head(2)
```

```
WARNING:root:Requests made without an app_token will be subject to strict throttling limits.
```

Out[612]:

analysis_neighborhood	cad_number	...	latitude	longitude	point	police_district	report_datetime	report_type_code	report_
Financial District/South Beach	183360210	...	37.78490829943045	-122.40479506275997	{'latitude': '37.78490829943', 'longitude': '-...	Southern	2018-12-02T01:56:00.000	II	
Tenderloin	183353564	...	37.78640961281089	-122.40803623744476	{'latitude': '37.78640961281', 'longitude': '-...	Central	2018-12-01T21:18:00.000	II	

Fig. 4. Data Access - Example

```
import org.neo4j.spark._
import org.apache.spark.graphx._
import org.apache.spark.graphx.lib._

//Neo4j object created from SparkContext
val neo = Neo4j(sc)

//Graphquery i Cypher for Data Loading
val graphQuery = """MATCH (n)-[r]->(m)
RETURN id(n) as source,
id(m) as target,
type(r) as value
SKIP {_skip} LIMIT {_limit}"""

//Graph Data loaded into spark
val graph: Graph[Long, String] =
neo.rels(graphQuery).partitions(10).batch(200).loadGraph

//Perform some algorithm triangle count

val results = //Triangle count
//Save results back into Neo4j DB

Neo4jGraph.saveGraph(sc, results, "Triangle count")
```

Fig. 5. Neo4J connector to Spark

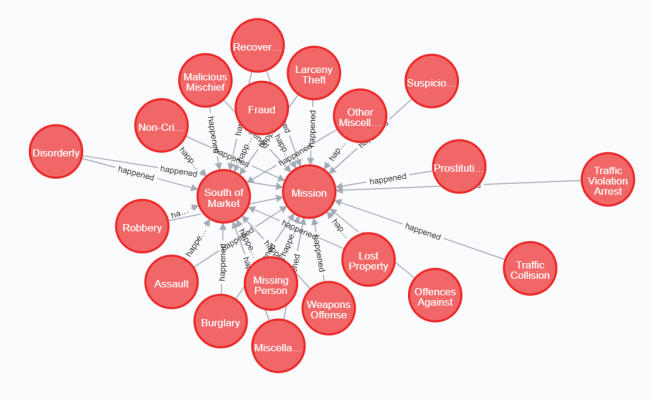


Fig. 6. Crime type and locations

D. *Spark*

Spark is chosen, as it has a lot of advantages over conventional coding, like the inherent advanced DAG execution framework engine where all the transformations to an RDD are stored as DAG and when an action is encountered the DAG is traversed and all the steps are executed(this is how lazy evaluation is done in spark) and in-memory computing. Thus it is 10x faster on disk and has compatibility with advanced high-level languages like Java, Python, Scala etc.

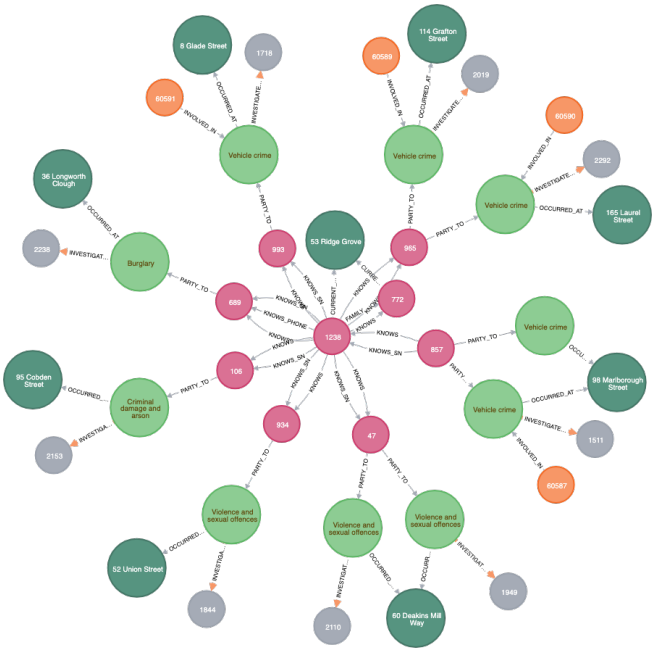


Fig. 7. Important crime locations

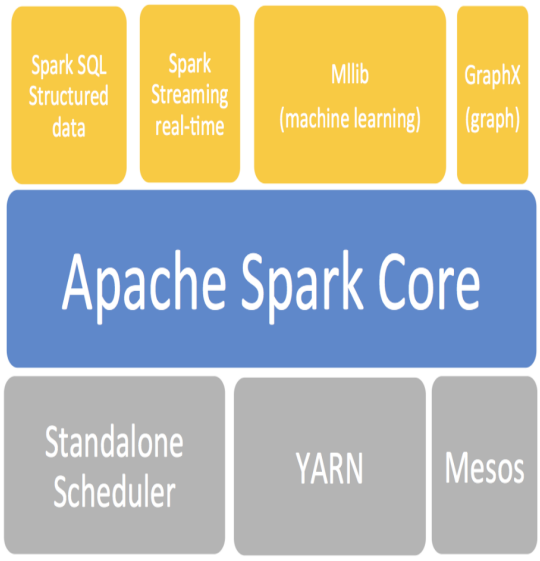


Fig. 8. Spark Stack

In addition to that Spark has around 80 high level operators to assist building parallel applications.

Spark also has stack of library like GraphX and MLlib for bringing the power of graphs and machine learning in our application.

E. Machine Learning

The whole ML pipeline is automated to run sequentially using programming. We form tokenized words from the sentences in every articles. We later remove the stop words. Following this TF and IDF for all the articles. The train set is from Mar 2013 to May 2018 and the test set is from May 2018 to current date.

*1) **Models**:* There are three models built for this demonstration, namely Random Forest Classification, Naive Bayes and Logistic regression.

```
from pyspark.ml.feature import RegexTokenizer, StopWordsRemover, CountVectorizer
from pyspark.ml.classification import LogisticRegression

regexTokenizer = RegexTokenizer(inputCol="description", outputCol="listofwords", pattern="\\W")

add_stopwords = ["http","https","the","in","of","if","to","for"]
stopwordsRemover = StopWordsRemover(inputCol="listofwords", outputCol="filteredword").setStopWords(add_stopwords)

countVectors = CountVectorizer(inputCol="filteredword", outputCol="features", vocabSize=1000, minDF=5)
```

Fig. 9. Feature Extraction sample

*2) **Feature Extraction**:* We have multitudes of words after filtering stop words and tokenization. We consider the most important words by generating the Term frequency for the words. We convert words into fixed length feature vectors called Hashing Term Frequency, where hashing is done for TF calculation. Then we do Inverse Document Frequency (IDF), where each column created by Hashing Term frequency.

*3) **Train Test**:* For the data we split train and test for the three models in ratio of 75 is to 25 percent. The models are trained for the three models and tested upon unknown data set. The label predicted is compared with the actual labels to predict accuracy of the system for the multi-class classifier models.

*4) **Pipeline**:* Figure 9 shows the steps of regexTokenization, stopword/special character filtering and Term frequency count for the model.

1) regexTokenizer: Tokenization using reg expressions
2) stopwordsRemover: Remove stopword/spl characters
3) countVectors: Count vectors TF and IDF
4) stringIndexer: Encodes label in strings to indices

F. Graph Algorithms

Fig. 10. Triangle Count

*1) **Triangle Count**:* A triangle is formed when a node has two adjacent nodes that are connected, ie. an edge exists

between the two adjacent nodes. Spark GraphX implements triangle counting algorithm by determining the number of triangle passing through each node. Thus this method is a kind of clustering of connected nodes. We can analyze incidents of such triangles to identify criminal gangs or suspect groups. From figure 10, we observe that the most prominent nodes in the overall graph are those pink nodes. Those 6 pink nodes are linked to almost 25 type of crimes and all 6 pink nodes are interconnected. So the inference might be that those 6 pink nodes can constitute a gang, and are connected to a variety of crime.

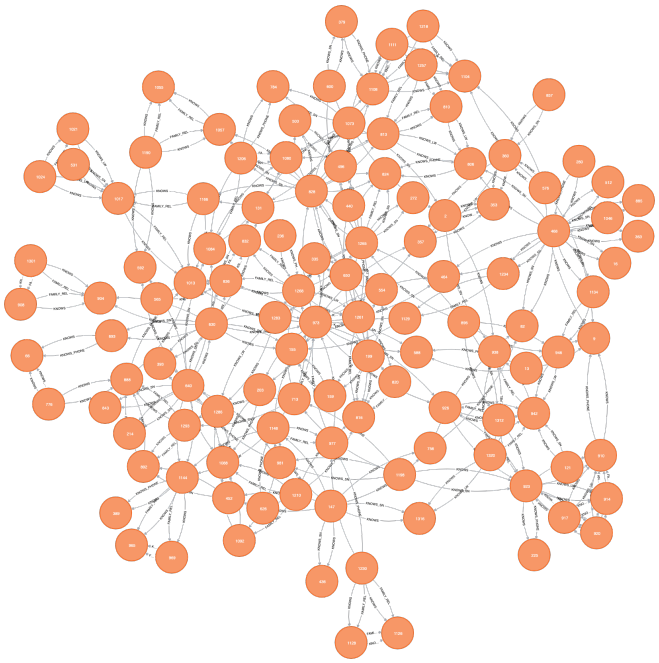


Fig. 11. Betweenness Centrality

*2) **Betweenness Centrality**:* The most important nodes in the graph can be identified using the betweenness algorithm. This measures the centrality of the overall graph by identifying the nodes that is located on the shortest path between multiple nodes. We can therefore identify the most important node, which sits on several clusters. In our use case we can identify the prevalent crimes in a neighbourhood by determining the shortest path between the crime nodes.From figure 11, we observe that there exists a lot of nodes which are connected to many other nodes in the graph.

IV. Outcome and Visualization

A. *Data Visualization EDA*

We visualize the data to understand how reliable our data is. Figure 12 provides an overview on the type of crime incidents, wherein we can notice that the most commonly committed crime is Theft. Also to show the spread of events in terms of duration, we plot the dates at which the event has occurred. From the figure 14, it is evident that most of the incidents has occurred in August and July. Figure 15 provides an more specific crime type and here we observe that rather than theft, assault is the most prevalent issue.

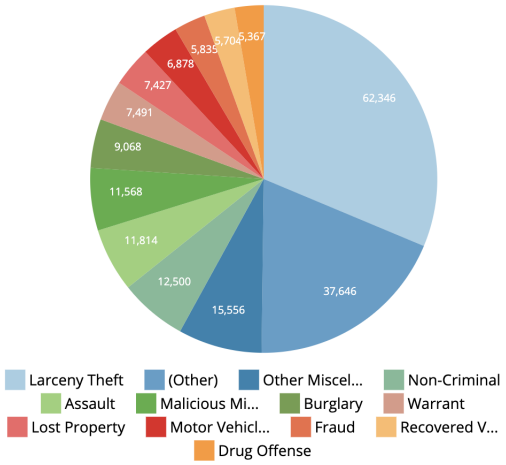


Fig. 12. Type of crime

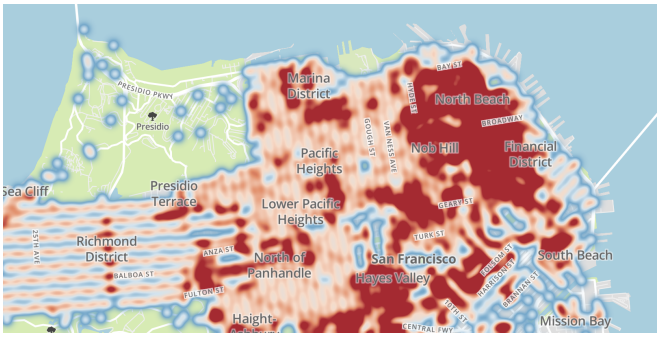


Fig. 13. Geolocation of crime

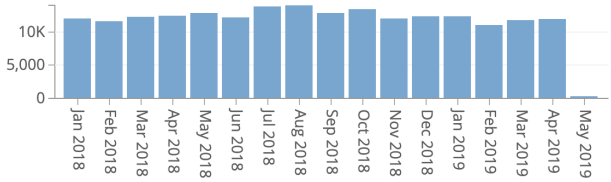


Fig. 14. Monthly Spread of crimes

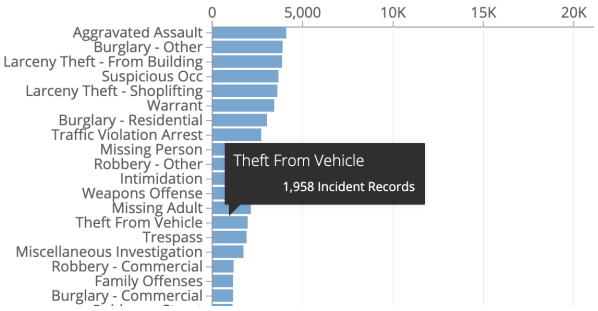


Fig. 15. Specificity of crime

B. ML model results

We ran our ML model locally, and have hosted it on Google colab to generate a sharable link mentioned in Reference 1.

https://colab.research.google.com/drive/1iLLXpdV43WkljTAqvQ32-hKedj6jb4ME

Figure 16 shows the count of incidents as a demonstration of how diverse and huge our dataset is. Also figure 17, shows Logistic regression model and its prediction and actual labels. Figure 18, shows the results of our model, which is 99 percent accurate.

Incident Category	count
Larceny Theft	62346
Other Miscellaneous	15556
Non-Criminal	12500
Assault	11814
Malicious Mischief	11568
Burglary	9068
Warrant	7491

Fig. 16. Category of incidents

Incident Description	Incident Category	probability	label	prediction
Theft, From Person, $200-$9...	Larceny Theft	[0.9352475281644416,0.00562...	0.0	0.0
Theft, From Person, $200-$9...	Larceny Theft	[0.9352475281644416,0.00562...	0.0	0.0
Theft, From Person, $200-$9...	Larceny Theft	[0.9352475281644416,0.00562...	0.0	0.0
Theft, From Person, $200-$9...	Larceny Theft	[0.9352475281644416,0.00562...	0.0	0.0
Theft, From Person, $200-$9...	Larceny Theft	[0.9352475281644416,0.00562...	0.0	0.0
Theft, From Person, $200-$9...	Larceny Theft	[0.9352475281644416,0.00562...	0.0	0.0
Theft, From Person, $200-$9...	Larceny Theft	[0.9352475281644416,0.00562...	0.0	0.0
Theft, From Person, $200-$9...	Larceny Theft	[0.9352475281644416,0.00562...	0.0	0.0
Theft, From Person, $200-$9...	Larceny Theft	[0.9352475281644416,0.00562...	0.0	0.0
Theft, From Person, $200-$9...	Larceny Theft	[0.9352475281644416,0.00562...	0.0	0.0

only showing top 10 rows

Fig. 17. ML result - prediction vs actual

```
metrics = MulticlassMetrics(predictionAndLabels)

# Overall statistics
precision = metrics.precision()
recall = metrics.recall()
f1Score = metrics.fMeasure()
print("Summary Stats")
print("Precision = %s" % precision)
print("Recall = %s" % recall)
print("F1 Score = %s" % f1Score)

```

```
Summary Stats
Precision = 0.9994785709479916
Recall = 0.9994785709479916
F1 Score = 0.9994785709479916
```

Fig. 18. Results of ML model prediction

C. Graph models script

There is no latest Python API for Graphx,so scala API is used for the implementation of the algorithms. From figure 19 and 20, we find the code snippet for the Graph algorithms of Triangle count and betweenness centrality respectively. In the spark Graphx triangle count algorithm, the Triangle Count object calculates the number of triangles that passes through each vertex in the graph which provides a measure of the clustering of the data. Here a vertex belongs in a triangle if it has two adjacent vertices with an edge that goes between them. In order to run triangle count,the data should should have edges in canonical orientation and the graph partitioned using Graph.partitionBy. Spark doesn't provide inbuilt algorithm for betweenness centrality,we use the package spark-betweeness for k Betweenness Centrality (kBC) algorithm for Spark using Graphx. The code snippet shows the usage of the algorithm from loading data and calling the respective function.

```
import org.apache.spark.graphx.{GraphLoader, PartitionStrategy}

// Load the edges in canonical order and partition the graph for triangle count
val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt", true)
  .partitionBy(PartitionStrategy.RandomVertexCut)
// Find the triangle count for each vertex
val triCounts = graph.triangleCount().vertices
// Join the triangle counts with the usernames
val users = sc.textFile("data/graphx/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val triCountByUsername = users.join(triCounts).map { case (id, (username, tc)) =>
  (username, tc)
}
// Print the result
println(triCountByUsername.collect().mkString("\n"))
```

Fig. 19. Triangle Count algorithm

Scala API

```
// import needed for the .avro method to be added
import com.centrality.kBC.KBetweenness

val graph = Graph(users, relationships, defaultUser)

// Run kBC to get result graph
val kBCGraph =
      KBetweenness.run(graph, k)
```

Fig. 20. Betweenness centrality algorithm

V. Summary

A. Summary - Learning outcome

The following points surmise our overall learning outcome in this project.

1) The importance of size and diversity in data. Data is of prime importance, as the overall system depends on it.
2) Building an automated data retrieval system which can access the recent articles from the datasource
3) Neo4J database - introduction, how to construct graphs.
4) Cypher query language - Queries to be run on Neo4J database.
5) Installation of spark - hurdles and the way we overcame it
6) Feature extraction - Maximizing data as 6 columns from 2 columns in addition to cross-fold validation
7) Importance of TF-IDF for word prominence detection
8) To run ML on spark for Naive, Logistic and Random forest classifier models
9) Graph algorithms and their implementation in Neo4J/Scala API.

10) Finding the EDA provided by SFData and tweaking it to satisfy our requirement

B. *Summary - Future work*

1) Housing the data in S3
2) Connecting Neo4J to Spark directly
3) Involving more complex ML models (Apart from tree models). Using ensemble (voting/bagging/boosting) in the future
4) Finding more relationship in the data
5) Publishing Google Datastudio and build a standalone application
6) Using real-time data from sources like Twitter, FB, Google news, RSS, NYTimes etc (Reliability/subjectivity is a concern)

REFERENCES

[1] Our ML Code *https://colab.research.google.com/drive/1iLLXpdV43WkljTAqvQ32-hKedj6jb4ME*.
[2] Spark Intro *https://runawayhorse001.github.io/LearningApacheSpark/why.html*.
[3] Spark installation *https://medium.com/@dvainrub/how-to-install-apache-spark-2-x-in-your-pc-e2047246ffc3*.
[4] API For SF data *https://dev.socrata.com/foundry/data.sfgov.org/wg3w-h783*
[5] SF Data - Train set *https://data.sfgov.org/Public-Safety/Police-Department-Incident-Reports-Historical-2003/tmnf-yvry/data*
[6] SF Data - Test set *https://data.sfgov.org/Public-Safety/Police-Department-Incident-Reports-2018-to-Present/wg3w-h783*
[7] SF Data - visualization *https://data.sfgov.org/d/wg3w-h783/visualization*
[8] Coursera - learning basics *https://www.coursera.org/learn/big-data-graph-analytics/lecture/JFZJf/hands-on-getting-started-with-neo4j*
[9] Graphx intro *https://spark.apache.org/docs/latest/graphx-programming-guide.html*
[10] Neo4j-Spark intro *https://medium.com/data-science-school/practical-apache-spark-in-10-minutes-part-7-graphx-and-neo4j-b6b01cffa4fd*
[11] Pyspark classification intro *https://towardsdatascience.com/multi-class-text-classification-with-pyspark-7d78d022ed35*
[12] *https://runawayhorse001.github.io/LearningApacheSpark/classification.html*