Analysis and Prediction of 911 Calls based on Location using Spark Big Data Platform

Project Report for Course IS 789 (Big Data Fundamentals and Techniques), Fall 2019

Team 1: Ketki Deshpande, Shruti Pandey, Sukhada Deshpande

Faculty Mentor: Jianwu Wang
Department of Information Systems, University of Maryland, Baltimore County

Abstract

Proper management of critical resources like Police Force and Ambulance Services is the key to establish peace quickly in times of crisis. When a police district receives a 911 call, quick response can be the difference between quiet handling and full riots in any area. Through this project, we have tried to determine frequent patterns for establishing association between day and time of the week, police district-based location and the reason for the call. We have also tried to predict the number of calls from a particular location (using longitude and latitude data). This data will help us manage police resources and put them to apt use as and when required. We have used Spark based algorithm known as FP-Growth for finding the frequent patterns in the calls and a couple of Regression algorithms Decision Tree and Random Forest for the prediction of calls based on location. Results show that weekend evenings are the busiest time for the emergency services, as most of the calls are made in the evenings of Friday, Saturday and Sunday. Also, Northwestern, Southwestern and Northeastern Police Districts get most of the calls in the evening. Based on the existing training data, we were able to predict new calls for a particular location.

1. Introduction

The crime rates have been increasing in recent times and so is the volume of 911 calls made. In such circumstance, a quick response to any 911 call is crucial. Time is of essence and an effective response is dependent on many factors including but not limited to the availability of police force and emergency vehicles. Keeping this in mind, it becomes critical to quickly and correctly process any 911 calls made and be ready for any kind of situations.

This can be achieved by finding out patterns between time and day of the week and the type of calls made. Also predicting what location might have the most calls using past data can be very effective. Since accuracy of the algorithm and speed of analysis are important factors, it is essential to use the right processing technique to empower our forces with the correct information in anticipation of any situation that might need immediate attention. For the purposes of this experimental project, the 911 records for Baltimore county have been used. This record set is publicly available and updated multiple times daily.

This report focuses on establishing patterns using the FP Growth algorithm. Multiple patterns were established between the calls and time of a day and day of the week. There is also an attempt to predict 911 calls from a given location based on latitude and longitude using Regression techniques. Solving this critical problem of management of resources can be made a lot simpler by using the best available technology in the most efficient way which is suitable for

our dataset. From the selection of algorithm to managing the number of executors and cores used to perform the calculations, these decisions will help us in providing an accurate estimate of how to distribute our resources to better handle emergency or non-emergency 911 calls at any point in time and from any location.
The later sections of this report talk in detail about the FP Growth algorithm and why it was a better choice for the Baltimore county 911 calls dataset instead of the more popular Apriori algorithm. Further arguments are made in comparison of decision trees versus the ensemble Random Forest Method which gives more accurate results than the former and their performance with respect to this data. To conclude, the effect of number of cores and executors on the execution time of an algorithm is scrutinized. This project was run on the UMBC High Performance Computing Facility (HPCF) provided by UMBC and all calculations and algorithm executions were made over the available Hadoop cluster. Spark was used for all implementations and that also is part of the reason for why one algorithm was more favored over another. This will be discussed in further detail in the upcoming sections.

2. Dataset

For understanding the call types, and predicting the location of any 911 calls, the publicly available dataset for 911 calls in Baltimore county was used. It was downloaded from Baltimore police department's website [1]. This dataset includes both emergency and non-emergency calls and consists of about 6.7 million records and 17 attributes at the time it was downloaded. The dataset is updated multiple times in a day.

2.1 Data Selection

For project purposes, only the data from January 1, 2019 was used in all the executions. The size of the final dataset came to about 1 million rows which translated to 130MB approximately, making it a big data dataset. To effectively predict the time of call and mine the patterns, the datatype of 'CallDateTime' column was changed from String to 'DateTime'. This was further broken down into Date and Time separately to mine the pattern between Day of the Week, Time of the Day (Morning, Afternoon, Evening, Night) and made in a format required by respective algorithms separately. Since all columns were not useful in our executions for predictions, the following columns were dropped-
RecordId, District, PolicePost, CounsilDistrict, SheriffDistrict, Community_statistical_area, Census_Tracts, VRIZones
This sliced dataset was loaded into HDFS environment made available by UMBC HPCF which is accessible by the Spark Programs.

2.2 Exploratory Data Analysis

Before making any predictions, the data was analyzed and explored to answer some common questions like the most common reason for calling 911, zip-codes making the most frequent 911 calls and the number of non-emergency calls made. This exploratory analysis enables trying to look for patterns easier and also provides a benchmark for what locations are more probable of having emergencies requiring immediate response.
The data exploration was performed by querying the data to get certain kind of information. SQL queries were written which extracted the required data from the Temporary View which was created on the Dataframe. Below command created the TempView required to query data directly using SQL:

```
df.createOrReplaceTempView("data_911")
```

It was important to find out what is the most common reason to call emergency number, and above TempView was queried for the same using below query.

```
sql("select Description, count(Description) from data_911 group by
Description order by count(Description) desc").show()
```

Below Bar Graph shows the output that above query generated.

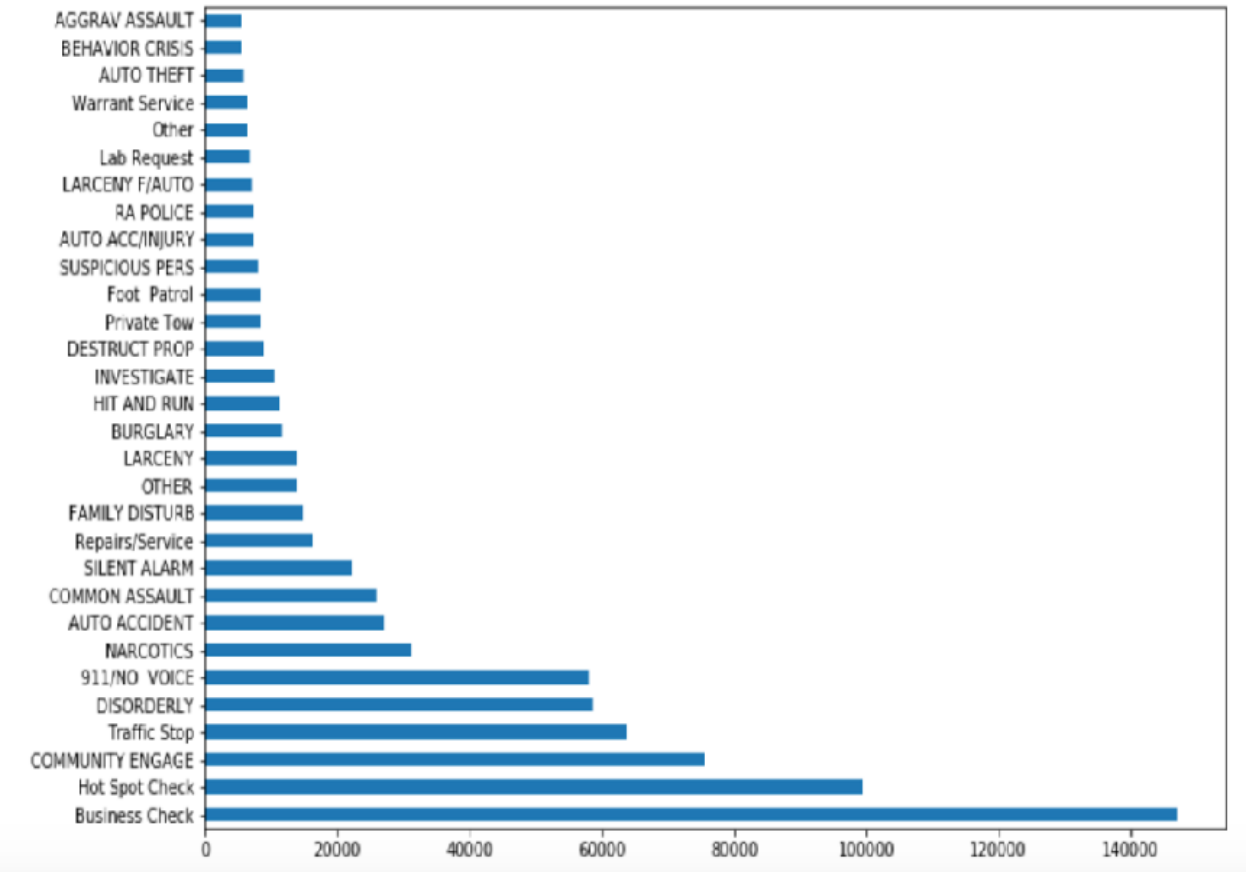


Figure 1: Plot of Number of 911 Calls per Description

The output shows that most common few reasons for calling emergency number is actually not an emergency, the reasons are categorized as non-emergency. When filtered the data to drop 'non-emergency' category and again queried for the most common reason, the output is represented by below Bar Graph:

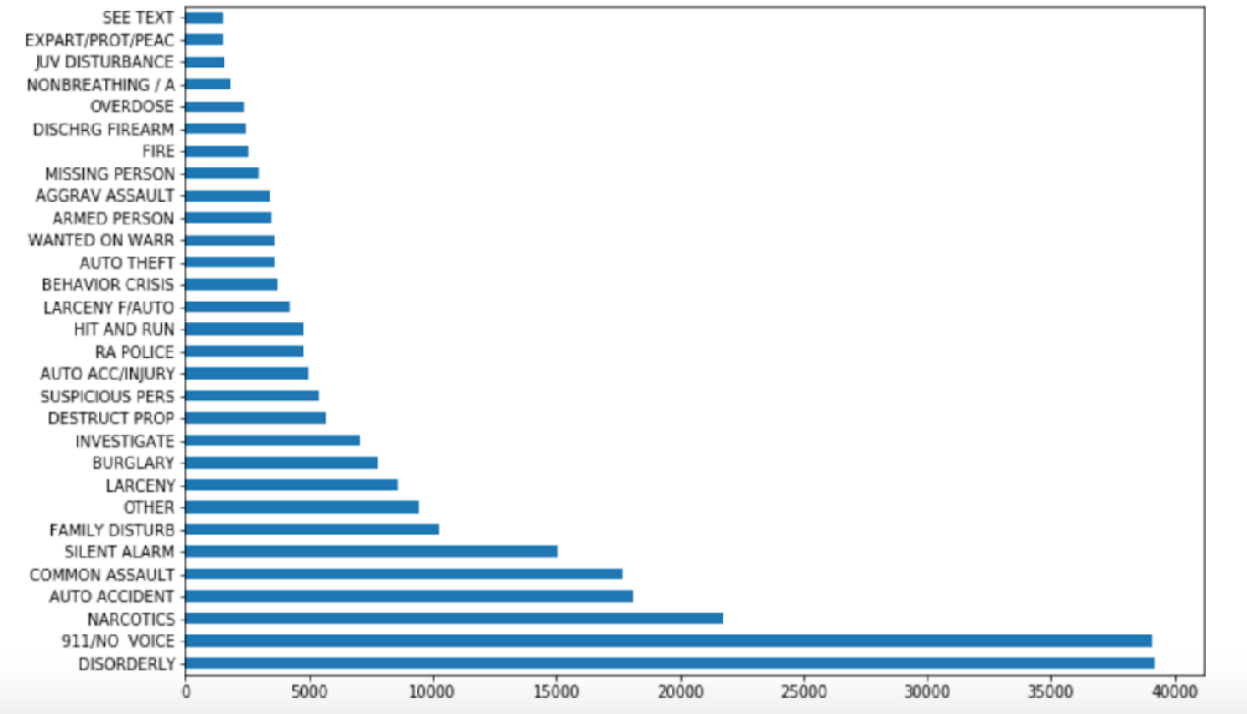


Figure 2: Plot of Number of 911 Calls per Description (when Non-Emergency category was excluded)

To get to know the count of 'non-emergency' calls that were made to 911 emergency number, below query was used:

```
sql("Select Priority, count(Priority) from data_911 where Priority=='Non-
Emergency' group by Priority").show()
```

The count came out to be 513407 which is almost over 50% of the total data that was selected for the purpose of this project. It was evident that half of the total calls that are made to emergency services are not for the emergency purposes. This information was useful in finding the Frequent Patterns in a given data. If all of data was used then most of Frequent Patterns were generated from the non-emergency section of the data with relations explaining reason of the call to its Priority. Knowing the count of 'non-emergency' calls was useful in limiting what data to use for finding Frequent Patterns and ultimately finding the correct patterns that were useful and interesting both.

The data was also queried to find out from which Zip Code the emergency number received most calls. Below query was used to find out this information:

```
sql("Select ZipCode, count(ZipCode) as ct from data_911 group by ZipCode
order by ct desc").show()
```

Below Bar Graph shows the output of the above query:

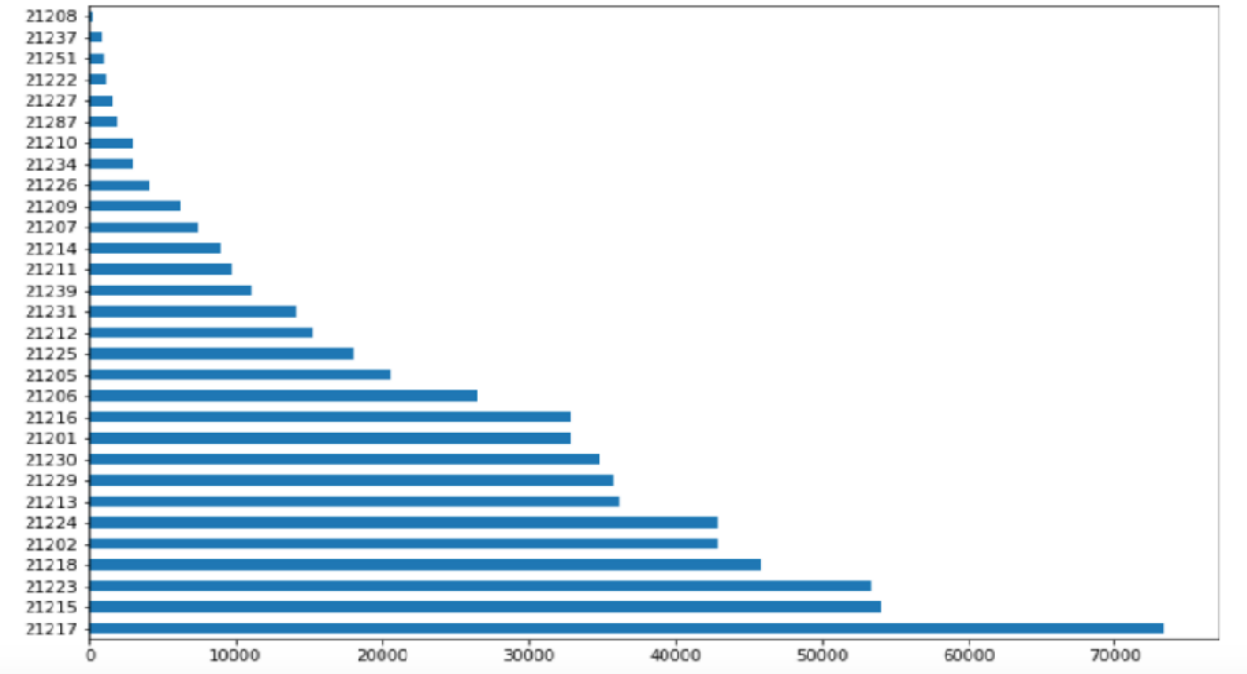


Figure 3: Plot of Number of 911 Calls per Zip-code

As this project aims to predict number of calls from a particular location, knowing which Zip Code gets the maximum calls was an important information to get a general idea about most busy zones for emergency services.

3 Mining Frequent Patterns from Data

Although most common use of Frequent Pattern Mining Algorithm [2] is for the 'market-basket analysis' [3], it is not limited only to this analysis. It might be interesting to find out Frequent Patterns in many other cases where data needs to be analyzed in a way to find out which two (or more) things occur together most frequently. Frequent Pattern Mining is a process to analyze the data with the aim to discover relations between the different attributes or associations between them. This project aims to find out frequent patterns in Description of a Call (reason for calling 911), Police District (used for location), Day of the Week and Time of the Day. It was a very useful information to know for example most of the calls for 'Auto Accident' were made in the evening. Identifying such patterns is useful in predicting where the police force might be needed most at a given time and hence help in better resource management.

FP-Growth using Spark

FP-Growth algorithm, where 'FP' stands for 'Frequent Patterns', was selected for the task of mining frequent patterns from the data over the popular algorithm Apriori [4]. The main difference between the Apriori and FP-Growth [5] is that FP-Growth creates FP-Tree, a suffix-tree structure whereas Apriori generates increasing number of candidate item-sets for mining frequent patterns. Using tree structure, a large amount of data can be compressed into smaller

data structure and the number of repeated database scans that Apriori performs can be avoided. The memory and time requirements for FP-Growth are also reduced as Apriori algorithm. So, considering the amount of data that was processed in this project for finding patterns in a data, a faster algorithm like FP-Growth was selected. It is designed to work best in distributed environment.
Spark uses parallel version of FP-Growth algorithm known as PFP [6]. PFP distributes the work of FP-tree for implementing the algorithm in parallel. The work distribution is based on transaction suffixes and hence this algorithm is more scalable in distributed environment than single-machine implementation. Spark has both RDD based and Dataframe based APIs for using FP-Growth algorithm. This project uses both APIs, and compare their performances on the same input data.
Spark's Dataframe based API [7] takes three hyperparameters which are explained below:

- `minSupport` – It is minimum support (minimum frequency) that is required by an itemset to be identified as frequent.
- `minConfidence` – It is a minimum confidence (minimum probability that the association rule holds true) for generating association rules. This parameter is only used while generating association rules and not used for mining frequent patterns.
- `numPartitions` – It is total number of partitions used for distribution of work.

The model created using this API outputs frequent itemset, association rules and also generates predictions on the input data using the generated association rules.
Spark's RDD based API uses two of the above three hyperparameters which are `minSupport` and `numPartitions`. The main difference apart from the fact that Dataframe based API uses Dataframe as input and RDD based API uses RDD as input, RDD based API does not have `minConfidence` as hyperparameter and there is no direct method or function to generate association rules using Python. Java and Scala APIs do have a function for generating association rules.

4 Regression for Predicting Count of Emergency Calls

Regression is a supervised learning method that can be primarily used for predicting and forecasting a range of numeric values from the given dataset. Since it is a supervised learning method, the training dataset mentioning the predicting class (the label which we are going to predict) need to be provided to the model in advance. For the purposes of this project, regression has been used to predict the number of 911 calls received from a location (considering latitude and longitude). Predicting 911 calls based on the location help us manage the police force resources of that particular vicinity and put them to apt use as and when required. To predict the number of 911 calls from a geo-location, regression task has been implemented using the most used machine learning algorithms: Decision Tree algorithm and the Random Forest algorithm.

4.1 Decision Tree

Decision tree is a popular method for machine learning tasks of classification and regression. Decision tress are widely used since they are intuitive, easy to interpret, does not require scaling of data and able to capture non-linearities and feature interactions. [8] Decision tree is a flowchart-like structure that includes a root node, branches and leaf nodes. Each internal node

represents a “test” on an attribute, each branch represents the outcome of the test, and each leaf node represents a class label. [9] Decision tree identifies the best attribute from the dataset and place that attribute at the root of the tree. Then split the training set into subsets which can be made in such a way that each subset contains data with the same value for an attribute. We need to continue the splitting step until we find the leaf nodes in all branches of the tree. [9]
Since decision trees are able to capture non-linearity, they are a good choice for this project. According to the Spark website, “decision tree is a greedy algorithm that performs a recursive binary partitioning of the feature space. The tree predicts the same label for each bottommost (leaf) partition. Each partition is chosen greedily by selecting the best split from a set of possible splits, in order to maximize the information gain at a tree node. In other words, the split chosen at each tree node is chosen from the set argmaxsIG(D,s) where IG(D,s) is the information gain when a split s is applied to a dataset D”.

4.2 Random Forest

Random forest is a Supervised Learning algorithm for classification and regression which uses ensemble learning method which combines the predictions from multiple machine learning algorithms together to make more accurate predictions than any individual model. In random forest multiple decision trees are combined to determine the final output rather than relying on single individual decision trees. The trees in random forests are run in parallel. There is no interaction between these trees while building the trees. [10] Random forest is one of the most accurate learning algorithms which runs efficiently on large datasets.
In random forest method, sample data points are taken repeatedly form the training dataset so that each data point is having equal probability of getting selected and all the samples have same size as the original training data set. A random forest regressor model is trained at each bootstrap sample and a prediction is recorded for each sample. Now the ensemble prediction is calculated by averaging the predictions of the above trees producing the final prediction. [11]

5 Data Pre-processing

Data preprocessing is one of the important steps in machine learning to build an efficient and effective model. The dataset contains of about 6.7 million records and this amount of real-world data is always incomplete and consist of errors. Preprocessing transforms this raw data into an understandable format [12]

5.1 Data Pre-processing for FP-Growth

To enable the FP-Growth to determine patterns in the call data, the data was preprocessed and made available in a format that algorithm requires. An important factor that this project aims to predict is the time of call and the type of call. To enable this pattern finding, the Timestamp column was divided into two columns- day of week and time of day. The column was split such that each date was represented as the respective day of the week, for example Monday, Tuesday and so on. The time of day was divided into four different values: Morning, Afternoon, Evening and Night based on the time. The time 6 AM to less than 12 PM was classified as Morning, 12 PM to less than 4 PM was classified as Afternoon, 4 PM to less than 9 PM was classified as Evening and time from 9 PM to less than 6 PM, full overnight time was classified as Night. These two columns were the primary inputs in all the pattern analysis executions. In addition to

these two columns, the columns Priority, Description, IncidentLocation (specifies rough area location or street address), ZipCode, Neighborhood, PoliceDistrict (used for location analysis) are joined together, comma separated into one column. This data was saved on HDFS as a .txt. The reason for saving the preprocessed data as a .txt is that both the RDD-based and Dataframe-based APIs can read this data from the .txt right from HDFS.

5.2 Data Pre-processing for Regression

In the proposed analysis, we have performed few data preprocessing techniques to obtain our data in good structure and understandable format. The dataset used was the same as mentioned earlier, but as different methods need input in different forms, the data was converted to the required format. The primary step was to read the dataset as a .csv file from HDFS which was executed on UMBC's HPCF. The rows with null or missing values for the columns 'PoliceDistrict', 'CallNumber', 'Priority', 'Description', 'Zipcode', 'IncidentLocation', 'Neighborhood' was dropped. Since the project aims at predicting calls from a geolocation, any rows that did not have the geolocation values specified were also dropped. Furthermore, the timestamp attribute was split into columns for year, month, dayofmonth, dayofweek, dayofyear, hour, minute, weekofyear. This splitting helps in predicting calls at a given time of a day from a given location. Followed by this, all attributes were converted to numeric since we are performing regression analysis on the dataset. This data was grouped by latitude, longitude, month, dayofmonth to get the current count of calls on a given day from a location. The data frame was then converted into an RDD and saved in 'libsvm' format to be fed into the algorithm implementations. The label value selected for both the algorithms was the 'count of calls' and the features were latitude, longitude, month, dayofmonth.
For both the approaches- Decision Tree and Random Forest, the model was trained on 70% data and tested on the remaining 30%. Evaluation metrics considered were Root Mean Square Error and Mean Absolute Error.

6 Experiments

After analyzing and pre-processing the data, next step is to train a model and analyze the output it generates. For this project FP-Growth, Decision Tree and Random Forest algorithms were trained and their generated outputs were analyzed. Models were trained using different values of their hyperparameters to get the required output, an interesting pattern or required prediction. The code used for this project can be found at the GitHub[1] link.

6.1 Experiments for FP-Growth

Experiments are performed to extract the frequent patterns in a data using different values of hyperparameters. Different combination of input values (different column values) would also result in different patterns. So, the experiments are conducted using different column values from the data as well as for the different values of hyperparameters. It was observed that both the APIs result in the same output for same data and same values of hyperparameters. Below is the sample output (frequent patterns in a data) when used hyperparameter value '`minSupport = 0.1`' and

[1] https://github.com/ketkideshpande/911_call_analysis

input values from the columns Description (reason for call), PoliceDistrict (used for location), day of week and time of day.
Sample output frequent itemset from Dataframe API:

```
+---------------+------+
|items          |freq  |
+---------------+------+
|[Mon]          |61225 |
|[Northwestern] |45518 |
|[911/NO  VOICE]|57995 |
|[Morning]      |92226 |
|[Wed]          |62671 |
|[Southeastern] |52234 |
|[Sat]          |63586 |
|[Sun]          |57903 |
|[Thu]          |61507 |
|[Southwestern] |45898 |
|[Afternoon]    |96365 |
|[DISORDERLY]    |58519 |
|[Northeastern] |61078 |
|[Central]      |57982 |
|[Fri]          |65327 |
|[Tue]          |62180 |
|[Southern]     |51000 |
|[Evening]      |127123|
+---------------+------+
```

Sample output frequent itemset from RDD API:

```
FreqItemset(items=['Mon'], freq=61225)
FreqItemset(items=['Northwestern'], freq=45518)
FreqItemset(items=['911/NO  VOICE'], freq=57995)
FreqItemset(items=['Morning'], freq=92226)
FreqItemset(items=['Wed'], freq=62671)
FreqItemset(items=['Southeastern'], freq=52234)
FreqItemset(items=['Sun'], freq=57903)
FreqItemset(items=['Sat'], freq=63586)
FreqItemset(items=['Thu'], freq=61507)
FreqItemset(items=['Southwestern'], freq=45898)
FreqItemset(items=['DISORDERLY'], freq=58519)
FreqItemset(items=['Afternoon'], freq=96365)
FreqItemset(items=['Northeastern'], freq=61078)
FreqItemset(items=['Fri'], freq=65327)
FreqItemset(items=['Central'], freq=57982)
FreqItemset(items=['Southern'], freq=51000)
FreqItemset(items=['Tue'], freq=62180)
FreqItemset(items=['Evening'], freq=127123)
```

Format of the output may different but the sample frequent itemset that both Dataframe and RDD APIs generate is same. Below table represents the output association rules from Dataframe API for different values from input columns and hyperparameters. Table represents only a few interesting association rules that were generated from all the conducted experiments.

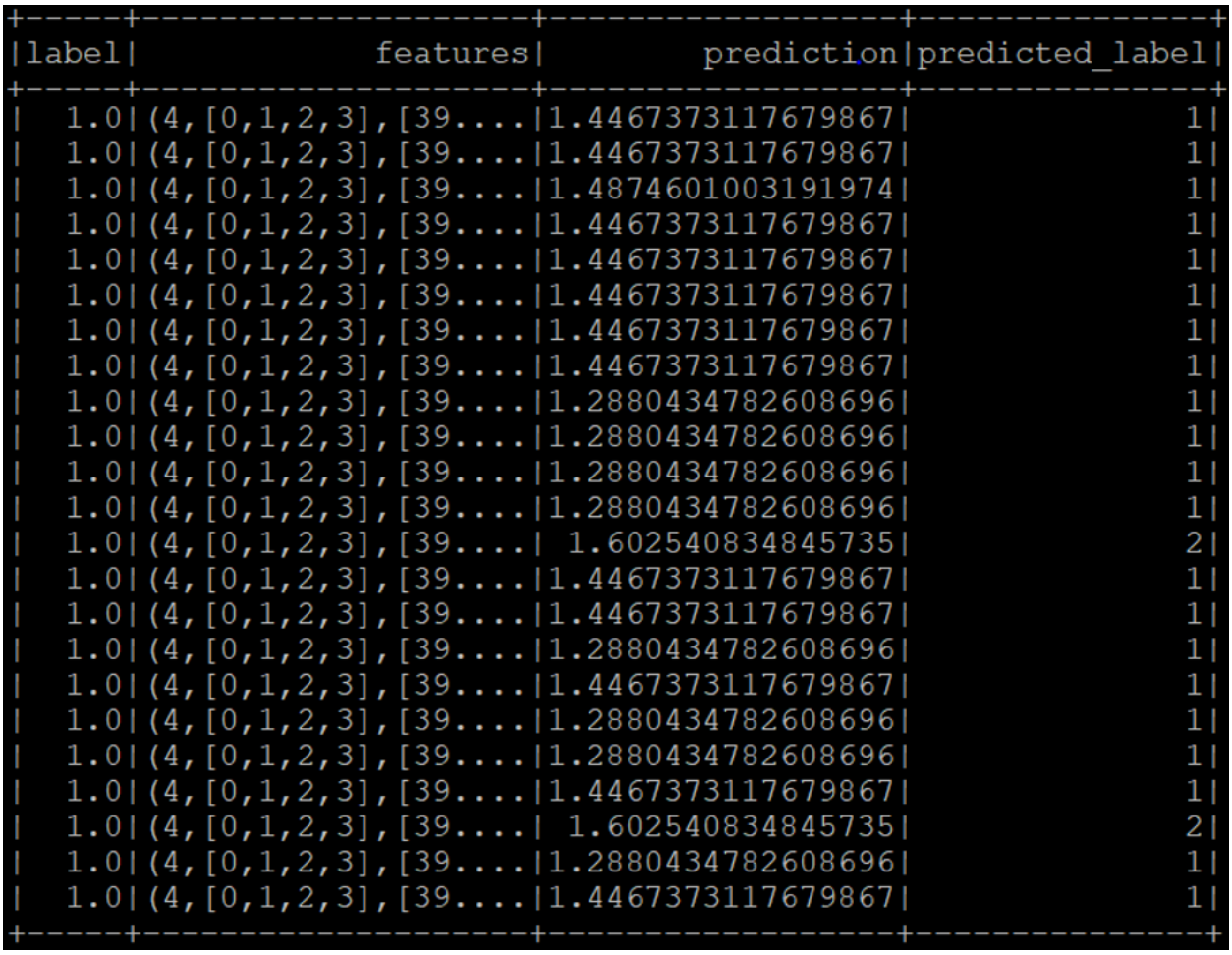
Columns Used	Hyperparameters	Result
Description, PoliceDistrict, Day of week, Time of day	minSupport=0.025 minConfidence=0.35	DISORDERLY -> Evening 0.4305282276373543 NARCOTICS -> Evening 0.4099707613161109 AUTO ACCIDENT -> Evening 0.36967832420938335
Police District, Day of week, Time of Day	minSupport=0.04 minConfidence=0.4	Northeastern -> Evening 0.42874946185394147 Southwestern -> Evening 0.41580984637716223 Northwestern -> Evening 0.40904667826497215
Description, Police District, day of week	minSupport=0.02 minConfidence = 0.15	DISORDERLY -> Central 0.15715970667168083 Central -> Fri 0.15290952364526922 Northeastern -> Sat 0.1514456923933331

Table 1: Association Rules with different inputs using FP-Growth Dataframe API

The above table mentions about the column values that were used as input to FP-Growth while creating a model and hyperparameters that were used. The 'Result' column of the above table explains about the association rules that were generated. For example, in the rule '`DISORDERLY -> Evening 0.4305282276373543`', the first word 'disorderly' is antecedent, second word 'evening' is consequent and the number represents confidence. Antecedent is a term (or item) that is frequently found in an input data and consequent is a term (or item) which is present in a transaction along with the term antecedent with a confidence given in the result.

6.2 Experiments for Decision Tree Regression

Below is the output for Decision tree algorithm showing the label, features (latitude, longitude, month, dayofmonth), prediction, and prediction_label. The label is the count of calls which we are predicting from a particular location, features are the input to the model from which we are predicting the label, prediction is our predicted output and predicted_label is just a standardize column to the nearest whole number.



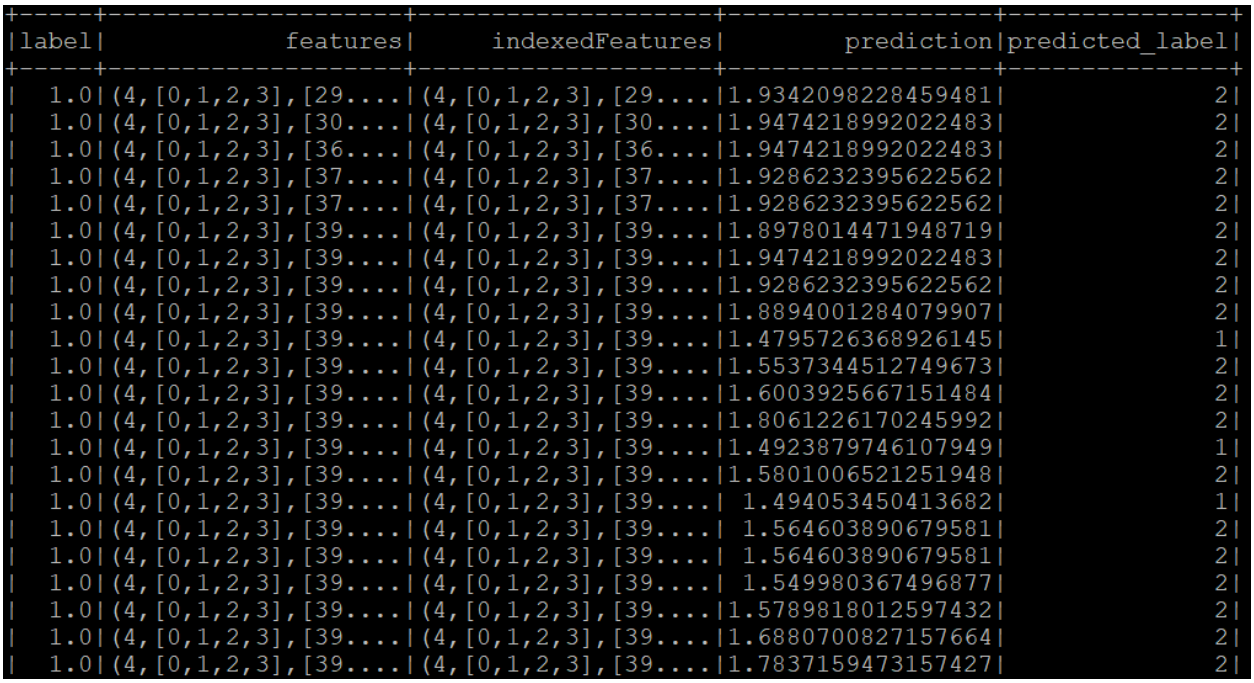
label	features	prediction	predicted_label
1.0	(4,[0,1,2,3],[39....	1.4467373117679867	1
1.0	(4,[0,1,2,3],[39....	1.4467373117679867	1
1.0	(4,[0,1,2,3],[39....	1.4874601003191974	1
1.0	(4,[0,1,2,3],[39....	1.4467373117679867	1
1.0	(4,[0,1,2,3],[39....	1.4467373117679867	1
1.0	(4,[0,1,2,3],[39....	1.4467373117679867	1
1.0	(4,[0,1,2,3],[39....	1.4467373117679867	1
1.0	(4,[0,1,2,3],[39....	1.4467373117679867	1
1.0	(4,[0,1,2,3],[39....	1.2880434782608696	1
1.0	(4,[0,1,2,3],[39....	1.2880434782608696	1
1.0	(4,[0,1,2,3],[39....	1.2880434782608696	1
1.0	(4,[0,1,2,3],[39....	1.2880434782608696	1
1.0	(4,[0,1,2,3],[39....	1.602540834845735	2
1.0	(4,[0,1,2,3],[39....	1.4467373117679867	1
1.0	(4,[0,1,2,3],[39....	1.4467373117679867	1
1.0	(4,[0,1,2,3],[39....	1.2880434782608696	1
1.0	(4,[0,1,2,3],[39....	1.4467373117679867	1
1.0	(4,[0,1,2,3],[39....	1.2880434782608696	1
1.0	(4,[0,1,2,3],[39....	1.2880434782608696	1
1.0	(4,[0,1,2,3],[39....	1.4467373117679867	1
1.0	(4,[0,1,2,3],[39....	1.602540834845735	2
1.0	(4,[0,1,2,3],[39....	1.2880434782608696	1
1.0	(4,[0,1,2,3],[39....	1.4467373117679867	1

Output for Decision Tree Regressor

From the above output, we can see that we are providing longitude, latitude, month, dayofmonth (in feature column) and label (1) and the model is correctly predicting the count of calls for some values in the dataset giving good accuracy. For example, in the first record, we are providing a label 1 to the model along with features and the model is correctly predicting the count of call as 1 for that location. The feature column represents a number of features available, encoded form of features which is required for the algorithm, and the actual values of the features. Here 4 is the number of features that is latitude, longitude, month and dayofmonth, [0,1,2,3] are the encoded form of features which is required for the algorithm and from 39 onwards are the actual features.

6.3 Experiments for Random Forest Regression

Below is the output of Random forest algorithm showing the label, features (latitude, longitude, month, dayofmonth), indexedFeatures, prediction, and prediction_label. We have applied the input to random forest algorithm in the same way as we have applied the input to decision tree algorithm. The label is the count of calls which we are predicting from a particular location, features are the input to the model from which we are predicting the label, prediction is our predicted output and predicted_label is just a standardize column to the nearest whole number.



label	features	indexedFeatures	prediction	predicted_label
1.0	(4,[0,1,2,3],[29....]	(4,[0,1,2,3],[29....]	1.9342098228459481	2
1.0	(4,[0,1,2,3],[30....]	(4,[0,1,2,3],[30....]	1.9474218992022483	2
1.0	(4,[0,1,2,3],[36....]	(4,[0,1,2,3],[36....]	1.9474218992022483	2
1.0	(4,[0,1,2,3],[37....]	(4,[0,1,2,3],[37....]	1.9286232395622562	2
1.0	(4,[0,1,2,3],[37....]	(4,[0,1,2,3],[37....]	1.9286232395622562	2
1.0	(4,[0,1,2,3],[39....]	(4,[0,1,2,3],[39....]	1.8978014471948719	2
1.0	(4,[0,1,2,3],[39....]	(4,[0,1,2,3],[39....]	1.9474218992022483	2
1.0	(4,[0,1,2,3],[39....]	(4,[0,1,2,3],[39....]	1.9286232395622562	2
1.0	(4,[0,1,2,3],[39....]	(4,[0,1,2,3],[39....]	1.8894001284079907	2
1.0	(4,[0,1,2,3],[39....]	(4,[0,1,2,3],[39....]	1.4795726368926145	1
1.0	(4,[0,1,2,3],[39....]	(4,[0,1,2,3],[39....]	1.5537344512749673	2
1.0	(4,[0,1,2,3],[39....]	(4,[0,1,2,3],[39....]	1.6003925667151484	2
1.0	(4,[0,1,2,3],[39....]	(4,[0,1,2,3],[39....]	1.8061226170245992	2
1.0	(4,[0,1,2,3],[39....]	(4,[0,1,2,3],[39....]	1.4923879746107949	1
1.0	(4,[0,1,2,3],[39....]	(4,[0,1,2,3],[39....]	1.5801006521251948	2
1.0	(4,[0,1,2,3],[39....]	(4,[0,1,2,3],[39....]	1.494053450413682	1
1.0	(4,[0,1,2,3],[39....]	(4,[0,1,2,3],[39....]	1.564603890679581	2
1.0	(4,[0,1,2,3],[39....]	(4,[0,1,2,3],[39....]	1.564603890679581	2
1.0	(4,[0,1,2,3],[39....]	(4,[0,1,2,3],[39....]	1.549980367496877	2
1.0	(4,[0,1,2,3],[39....]	(4,[0,1,2,3],[39....]	1.5789818012597432	2
1.0	(4,[0,1,2,3],[39....]	(4,[0,1,2,3],[39....]	1.6880700827157664	2
1.0	(4,[0,1,2,3],[39....]	(4,[0,1,2,3],[39....]	1.7837159473157427	2

Output for Random Forest Regressor

From the above table, we can say that the accuracy for the random forest algorithm is not good. The prediction is not as good as decision tree algorithm as we can see a lot of wrong predictions for some values in the dataset.
We have also calculated the root mean square error and mean absolute error for both random forest and decision tree algorithms to get the accuracy result and we got below result mentioned in the table.

Algorithms	Runtime	RMSE	MAE
Decision Tree	0m38.008s	0.0247515	1.16086
Random Forest	0m44.746s	2.06309	1.14838

Table 2: Comparison between Decision Tree and Random Forest approaches

From the above table, it is seen that we got a better runtime for decision tree algorithm as compared to random forest algorithm. Also, root mean square error (RMSE) for the decision tree is way better than the random forest which means the accuracy of decision tree is better than random forest algorithm. There is not much difference in the mean absolute error (MAE) for decision tree and random forest.
Usually random forest gives better result as it is an ensemble learning method. But here in our analysis we can see the random forest is not performing well as compared to decision tree algorithm.

7 Performance Evaluations

All the experiments were conducted on the High-Performance Computing Facility (HPCF) provided by UMBC. HPCF has a Big Data Cluster consisting of a Management Node and eight Worker Nodes. Majority of the computing tasks are performed on the Worker Nodes, each of which has two 18-core Intel Xeon Gold 6140 Skylake CPUs, for a total of 36 cores per node [13]. Each node has 384 GB of memory and 48 TB SATA hard disks and are connected together by an Ethernet connection with a 10 Gb/s speed. Users can directly use/ work on an Edge/Login Node which is the only Node accessible SSH/SCP from outside of the cluster. The UMBC AFS storage can be accessed via Big Data Edge/Login Node. Apart from that, Edge/Login Node has a local directory named /scratch which is shared between all users and has a capacity of 800 GB. Data transfer to HDFS can be originated from /scratch directory.

7.1 Performance Evaluation for FP-Growth

As the Big Data Cluster is used for performing experiments, the performance is evaluated based on different combinations of Executors, Cores and Memory. We can manually specify then number of Executors, Cores and Memory and find out which combination works best for our program, the one that gets us a minimum execution time. The FP-Growth APIs, both Dataframe based and RDD based, have a hyperparameter called '`numPartitions`' where we can specify how many partitions to use for distributing the work on. This hyperparameter has also been evaluated for the same input data.

The table below shows the performance of our program for different number of Executors, Cores and Memory for Dataframe based API as well as RDD based API of FP-Growth algorithm:

RDD API				Dataframe API			
Executors	Cores	Memory	Time	Executors	Cores	Memory	Time
10	18	100g	0m19.229s	10	18	100g	0m24.598s
10	24	10g	0m41.247s	10	24	10g	0m45.539s
10	36	10g	0m40.989s	10	36	10g	0m48.505s
10	36	100g	0m40.701s	10	36	100g	0m48.426s
10	8	10g	0m19.627s	10	8	10g	0m26.995s
10	4	10g	0m19.317s	10	4	10g	0m26.470s
10	4	100g	0m21.146s	10	4	100g	0m27.716s
2	**4**	**10g**	**0m15.466s**	2	4	10g	0m22.022s
1	4	10g	0m15.942s	**1**	**4**	**10g**	**0m21.265s**
4	4	10g	0m18.573s	4	4	10g	0m23.193s
6	12	100g	0m17.137s	6	12	100g	0m23.198s
8	12	100g	0m19.207s	8	12	100g	0m23.317s

Table 3: Execution times for Dataframe based and RDD based APIs with different combinations of executors, cores and memory

In the above table, the minimum execution time is highlighted. So, the general observation for the given input data is with a smaller number of executors, cores and memory the performance was improved. This might be due to the fact that input data is not big enough to actually use large number of cores and executors.

Below table explains the performance of Dataframe based and RDD based APIs for FP-Growth when the hyperparameter '`numPartitions`' is used with different values:

API	Number of Partitions			
	1	2	5	8
Dataframe based	0m23.775s	0m24.004s	0m24.755s	0m26.429s
RDD based	0m16.319s	0m16.864s	0m16.585s	0m17.551s

Table 4: Execution times for Dataframe based and RDD based APIs with different number of partitions value of hyperparameter

The observations show that performance is better when smaller number of partitions are used. Another interesting observation is that the RDD-based API performed much faster with the same amount of data as the Dataframe-based API. This contrasts with the general performance of RDD-based versus Dataframe-based APIs since the Dataframe-based APIs are known to perform better for most use cases.
One argument that can be made in view of these observations is that depending on the data, since the algorithm being run is not taking columns from the dataset but arrays of data that have been designed to mine patterns, it is more efficient for the RDD-based API.

7.2 Performance Evaluation for Regression

The table below shows the performance of our program for different number of Executors, Cores and Memory for Decision Tree algorithm and Random Forest algorithm:

Decision Tree				**Random Forest**			
Executor	**Core**	**Memory**	**Time**	**Executor**	**Core**	**Memory**	**Time**
10	18	100g	0m32.278s	10	18	100g	0m45.616s
10	24	10g	0m33.422s	4	18	100g	0m45.197s
10	36	10g	0m33.878s	4	36	100g	0m47.301s
10	36	100g	0m32.191s	4	10	100g	0m43.229s
10	8	10g	0m35.828s	4	10	50g	0m46.173s
10	4	10g	0m35.967s	10	10	10g	0m44.360s
10	4	100g	0m33.163s	4	4	100g	0m43.445s
2	4	10g	0m35.323s	10	10	100g	0m42.599s
1	4	10g	0m34.222s	20	10	100g	0m44.089s
4	36	100g	0m36.567s	25	25	100g	0m43.969s
6	24	20g	0m31.946s	**25**	**36**	**100g**	**0m42.513s**
6	**36**	**100g**	**0m31.570s**	36	36	100g	0m45.153s
4	36	100g	0m33.889s	20	36	100g	0m44.923s
8	36	50g	0m35.898s	30	36	100g	0m43.878s

Table 5: Execution times for Decision Tree and Random Forest with different combinations of executors, cores and memory

The observations show that performance is better when a greater number of cores and more memory is used. Another interesting observation is that the Decision Tree algorithm is faster than the Random Forest algorithm. This contrasts with the general concept that Random Forest is faster since it is an ensemble method.

8 Conclusion

FP Growth works faster with a smaller number of cores and executors. Reason could be that, FP Growth is a tree algorithm and each node require root node. So even when implemented a parallel version of FP Growth, it may require more time. Also, the data that was used FP Growth RDD API produces faster result with same amount of data compared with Dataframe API. Even though Random Forest is an ensemble method and generally gives higher accuracy, for our dataset we could get lower RMSE value by Decision Tree. Increasing or decreasing the counts of executors, cores and memory does not seem to have more effect on the time of execution but still with higher number of executors and cores, we could get minimum execution time.

Acknowledgement

The hardware in the UMBC High Performance Computing Facility (HPCF) is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS–0821258, CNS–1228778, and OAC–1726023) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See hpcf.umbc.edu for more information on HPCF and the projects using its resources.

The project team would like to thank Dr. Jianwu Wang for his guidance and mentorship. This project would not have been possible without the infrastructure provided by UMBC's High Performance Computing Facility (HPCF).

References

[1] B. P. Department, "911 Police Calls for Service," 2019. [Online]. Available: https://data.baltimorecity.gov/Public-Safety/911-Police-Calls-for-Service/xviu-ezkt. [Accessed 24 November 2019].

[2] A. C. Agarwal R.C., "Efficient Algorithms for Mining Long Patterns in Scientific Data Sets," *Grossman R.L., Kamath C., Kegelmeyer P., Kumar V., Namburu R.R. (eds) Data Mining for Scientific and Engineering Applications. Massive Computing, vol 2. Springer, Boston, MA,* vol. vol 2., 2001.

[3] A. R. Ltd., "Market Basket Analysis," 2019. [Online]. Available: https://www.albionresearch.com/data_mining/market_basket.php. [Accessed 12 December 2019].

[4] Wikipedia, "Apriori algorithm," 2019. [Online]. Available: https://en.wikipedia.org/wiki/Apriori_algorithm. [Accessed 11 December 2019].

[5] J. Han, J. Pei and Y. Yin, "Mining frequent patterns without candidate generation," *ACM SIGMOD Record,* vol. 29, no. 2, pp. 1-12, 2000.

[6] H. Li, Y. Wang, D. Zhang, M. Zhang and E. Y. Chang, "Pfp: parallel fp-growth for query recommendation," in *Proceedings of the 2008 ACM conference on Recommender systems*, Lausanne, Switzerland, 2008.

[7] A. Spark, "Frequent Pattern Mining," Apache, [Online]. Available: https://spark.apache.org/docs/latest/ml-frequent-pattern-mining.html. [Accessed 05 December 2019].

[8] "Spark 2.4.4 Documentation," [Online]. Available: https://spark.apache.org/docs/latest/ml-classification-regression.html#decision-trees.

[9] "Medium," [Online]. Available: https://medium.com/greyatom/decision-trees-a-simple-way-to-visualize-a-decision-dc506a403aeb.

[10] T. D. Science. [Online]. Available: https://towardsdatascience.com/random-forest-and-its-implementation-71824ced454f.

[11] GDCoder. [Online]. Available: https://gdcoder.com/random-forest-regressor-explained-in-depth/.

[12] "Towards Data Science," [Online]. Available: https://towardsdatascience.com/data-pre-processing-techniques-you-should-know-8954662716d6.

[13] U. o. M. B. County, "Description of the Big Data Cluster," UMBC, [Online]. Available: https://hpcf.umbc.edu/system-description-of-the-big-data-cluster/. [Accessed 10 December 2019].