Chicago Police Officer Scheduling System

Crime and traffic pattern detection, prediction and analysis using distributed data mining

Bjarte Botnevik
University of Stavanger, Norway
b.botnevik@stud.uis.no

Eirik Sakariassen
University of Stavanger, Norway
e.sakariassen@stud.uis.no

ABSTRACT

Crimes affects everyone in some way. Individuals that is directly affected by crimes may suffer psychologically and physically. Individuals who aren't directly affected may develop an increased fear of crime in certain cases, and the financial impact of crimes affects the society in different ways, such as higher insurance rates [11]. In order to help prevent crimes, we developed CPOSS (Chicago Police Officer Scheduling System), which is a system that forecasts and provides crime probabilities within different blocks of Chicago. This paper shows how we conducted exhaustive experiments on historical data from Chicago Police Department's CLEAR (Citizen Law Enforcement Analysis and Reporting) system, as well as how we fine tuned and combined different models into an overall system. The entire work was performed on distributed clusters, which enabled us to process large amounts of data fast. Finally, we demonstrate how CPOSS performs on future data by looking at an simulated live server.

KEYWORDS

Time series forecasting, Spark, Hadoop, MLlib

1 INTRODUCTION

It is crucial that the police is at the place of a crime in time. This could be to catch the bad guys, to mitigate the damages, or to even prevent the crime. The hardest part about this today is that the police has no way of knowing when or where the next crime is going to happen. Our goal is to make CPOSS as much of a help as possible for the police. So that they can prevent as many of the crimes as possible.

By predicting what type of crimes is going to be next can be very beneficial. The police forces can be prepared, and bring the fitting equipment to either prevent the crime, or make it as least serious as possible. Both preventing damages to the people involved, and to the police officers themselves.

Forecasting both traffic jams and traffic crashes can be helpful to provide the police with the shortest route to the crime as possible. The police can avoid those streets that are predicted to have high traffic, and avoid areas where there are predicted crashes. The police cars can also be placed around the areas so that it does not have to drive on a high traffic road to reach its destination.

1.0.0.1 Research Problem. This is not as straight forward as it might seem. Say that we get a system that works. There is a chance that CPOSS will predict that the majority of crimes will happen at the most violent areas of Chicago. Thus, CPOSS will suggest to put most of the police cars in and around those areas. The amount of crimes in these areas will most likely go down, but it is naive to think that the total amount of crimes will go down. What then might happen is that the crimes will happen in other areas, that before was seen as more safe. This is not desirable either.

1.0.0.2 Contribution Summary. This is the key contribution we did

- We used MapReduce jobs to find anomalies in the data.
- We used Spark DataFrames when working with the data.
- Prophet was used for forecasting time series in parallel across multiple clusters.
- We used MLlib for classifying crime types.
- We developed custom pipeline testers for ML tasks.
- Feature engineering was done to improve the classifying models.

2 BACKGROUND

In this section we explain the most important components of the project. We will provide enough information so that the reader can understand the concepts behind the project. The data sets will be presented and lastly some related works.

2.1 Time Series

A time series is a set of sequential observations. The set of observations can be any type of variable that changes over time. The sequence itself may vary between time series, for instance monthly, weekly or daily sequences of server traffic. These properties makes the time series useful for a wide range of applications which relies on how variables changes over time. Different fields may have different motivation for these application. Applications within geophysics and finance often use time series for forecasting, whereas applications within data mining and machine learning can use them for anomaly detection, clustering, forecasting and more.

2.2 Time Series Forecasting

Different characteristics of time series can be modeled and used for forecasting tasks. High performance forecasting systems is often derived from a complex analysis of the data, supported by domain expertise. We will thus briefly introduce only the key concepts used in this papers forecasting tasks.

The time series data is often decomposed into multiple components of interest.

Trends shows how the data decrease or increase over a longer period of time. Trend trajectories may have abrupt changes, separated by trend changepoints. Growth between these changepoints can be modelled by suitable models. For instance a *nonlinear saturating growth model* can model the crime occurrences in cases

Supervised by Jayachander Surbiryala.

Project in Data Intensive Systems (DAT500), IDE, UiS
2020.

where growth saturates at a carrying capacity, e.g. number of police patrols [10]. Such logistic growth models follows the form:

$$g(t) = \frac{C}{1 + e^{-k(t-m)}}, \tag{1}$$

where C is the fixed carrying capacity, k is the constant growth rate and m is an offset parameter. The police capacity might vary over time t and the growth of crime occurrences will probably not be constant, hence a time-varying capacity C(t) and rate adjustments $\boldsymbol{\delta} \in \mathbb{R}^S$ could be added to the model. The rate change occurring at changepoint $s_j, j = 1, .., S$ is defined by δ_j. The rate at time t is the base rate k plus all adjustments up until that time: $k + \boldsymbol{a}(t)^T \boldsymbol{\delta}$, where $\boldsymbol{a}(t) \in \{0, 1\}^S$ is such that:

$$\boldsymbol{a}(t) = \begin{cases} 1 & if \quad t \geq s_j \\ 0 & otherwise \end{cases}$$

The offset parameter m must be adjusted whenever k is adjusted in order to connect endpoints of segments. The adjustment parameter λ_j at changepoint j is defined by:

$$\lambda_j = \left(s_j - m - \sum_{l<j} \lambda_l \right) \left(1 - \frac{k + \sum_{l<j} \delta_l}{k + \sum_{l \leq j} \delta_l} \right)$$

Finally, all the information above can be combined into a final *piecewise logistic growth* model:

$$g(t) = \frac{C(t)}{1 + e^{-(k+\boldsymbol{a}(t)^T \boldsymbol{\delta})(t-(m+\boldsymbol{a}(t)^T \boldsymbol{\lambda}))}}, \tag{2}$$

Seasonality is referred to as periodic fluctuations that is repeated each season. A season can be longer periods like summer vacation and spring break, but also shorter periods like workdays and weekends. Fourier series can be used as a seasonality model since it is a periodic function of t. With P as a regular period, e.g. P=5 for workdays, a standard Fourier series

$$s(t) = \sum_{n=1}^{N} \left(a_n cos\left(\frac{2\pi nt}{P} \right) + b_n sin\left(\frac{2\pi nt}{P} \right) \right), \tag{3}$$

can approximate seasonal effects.

Holidays and Events may have big impact on the time series, year after year. They do often not follow a strict periodic pattern, and hence a seasonality model assuming smooth cycles wont work well. For instance, Thanksgiving is a national holiday in the US which occurs every year on the fourth Thursday of November. Given a set of past and future holidays, D_i, assume i is an independent holiday. The change that holiday i causes on the forecast is represented in a parameter k_i. By further adding an indicator function to tell whether time t is in holiday i, one end up with:

$$Z(t) = [\mathbf{1}(t \in D_1, ..., \mathbf{1}(t \in D_L)],$$

which is a matrix of regressors that can be used to incorporate holidays into a model:

$$h(t) = Z(t)\mathbf{k},$$

where $\mathbf{k}$ is a prior.

These components are usually visible through line plots, as shown in figure 1.

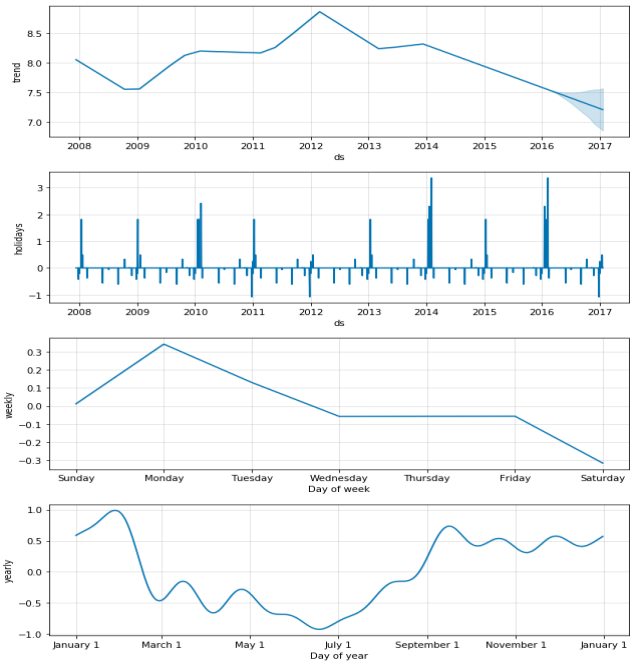


Figure 1: Decomposition of a time series

2.3 Anomaly Detection

The point of anomaly detection is to find outliers in the data. There are many reason why you would want to find these outliers. Anomalies can greatly influence the result of the analysis. The outliers may be errors in the data, or data that does not make much sense and is not helpful in the analysis. The common factor of these anomalies is that their data points deviate from the behavior of the normal data. Of course, some events may cause different behavior in data, but unexpected change in behavior is considered an anomaly.

2.3.1 Test Score. The way to find outliers is to check if the data point is outside the majority of data. This is done by running the equation on every data point. *x_i* is the data point. *med(X_n)* is the median of the values, and *IQRN(X_n)* is the interquartile range of the values

$$r_i = \frac{x_i - med(X_n)}{IQRN(X_n)},$$

In some cases it can be hard to determine whether or not the outliers are faulty data. All the test score is telling us is that the data is outside the majority of the data. If we check that the score is above three, meaning three standard deviations, it means that the data point is outside 99.7% of the data.

2.4 Chicago Crime Data set

The data set contains crimes from Chicago, Illinois. Chicago is known to be one of the most violent cities in the USA. Crimes happening from 2001 to present is stored and published by the city of Chicago [1].

2.5 Chicago Traffic Crashes Data set

Traffic crashes from Chicago is stored and published in a data set by the city of Chicago [8]. Data from 2015 to present is recorded and updated daily. 49 features describe each crash, and it is about 399 000 crashes.

2.6 Chicago Traffic Tracker Data set

Traffic has been tracked for Chicago from August 2011 to May 2018 [9]. This data set contains 19.6 million rows of traffic information, containing 5 features. The column containing the speed of the traffic is most interesting. It contains information about the speed at which the traffic is going, measured in miles per hour. The values are divided in to 0-9, 10-20, and 21 and above, corresponding to heavy, medium and free flowing traffic.

2.7 Related Works

Uber does a lot of time series forecasting, and is the most relevant related work we could find [3]. Uber tries to predict the amount of traffic they will have. They predict where the customers will be, how many there will be, and when the customers needs to be picked up. Predicting where a police car should be at a given point as corresponding with the problem that Uber is trying to solve.

They are also doing anomaly detection, to detect faulty data. One example is that they got lots of false positives rates during the holidays [6]. All though this might not be faulty data, the data might not be relevant. These are not like the typical Saturday night spike in activity, because this is expected behaviour.

There are some papers and works that have done classification on the type of crimes from the Chicago crimes data set [5] [7]. However, they have not done it the same way we have. The main difference is that they have the FBI code and crime description included in the training set. The best score that they achieve is 99.92%, which is a really good score. However, the FBI Code directly corresponds with type of crime. This information is clearly unavailable for a real time system and makes their classifier sort of useless. They do also include very detailed location and time data that would require highly advanced algorithms if they where to be generated for future predictions. Our model base all predictions on available time derivatives and district location only. We develop custom algorithms in order to advance the models based on these limitations.

3 PREPROCESSING

Before the development of good forecasting systems and classifiers, one has to explore, understand and convert the data in such a way that it can be used on real time applications. We performed thorough analysis on millions of crime records over many years, divided by internal geographical borders. The exploration enabled us to discover patterns and features of the data that further could be used to improve and understand the systems that was developed.

This section will go through how we performed different preprocessing tasks. We will discuss implementations using different frameworks, and present the most interesting findings in the analysis.

3.1 Trends and Growth

Time series generated by human actions tend to have seasonality effects. Therefore, trends and growth of crime occurrences was analyzed so we could provide more accurate seasonality information in a time series forecasting algorithm. We looked at hourly, daily, weekly and monthly data of the entire city of Chicago as well as each individual district.

We applied moving average smoothing on time series by using the *rangeBetween* function in the SparkSQL Window class on rows ordered by date. Such a smoothing technique enabled us to determine underlying trends. Figure 3 shows daily crime counts from 2001–2020 with and without a 7 day moving average. We can tell that the crime frequency has decreased over time and flattened out around 2015. There were clear yearly seasonality as well as big outliers a few days of each year.

Additional processing was applied on the original DataFrame so we could see how the daily count had changed over time for each district. In order to achieve that, we had to group the data by district, then apply counting aggregation, and in the end create a pivot table with each district as an own column. 30 day moving average was computed for each district before visualization. Most districts followed a descending growth similar as the overall plot, but some few districts experienced a recent rise of crimes.

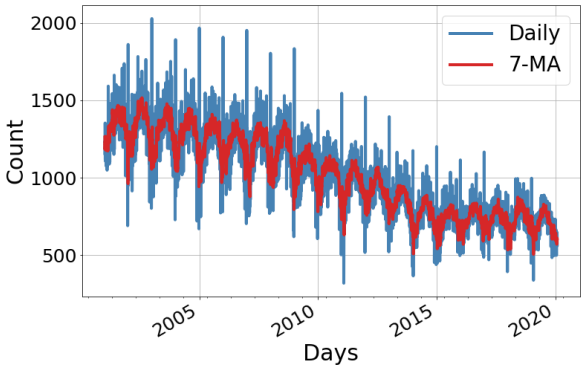
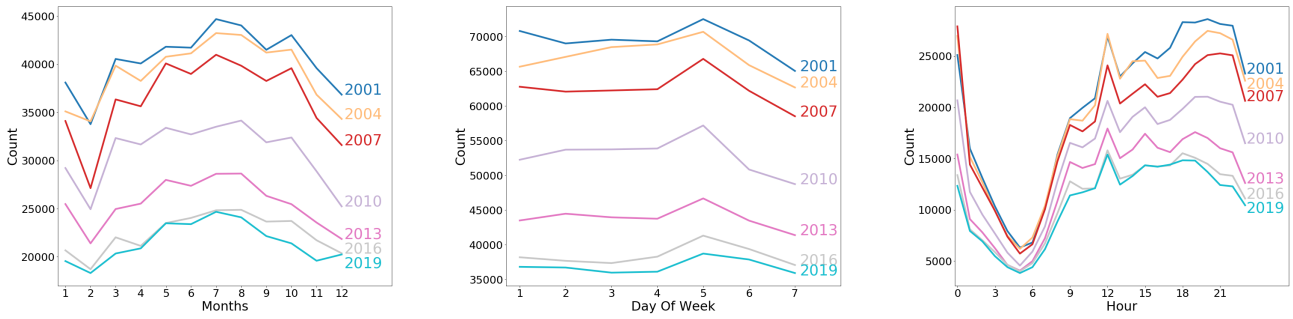


Figure 3: Daily crime count from 2001-2020

Further analysis involved hourly, weekly and monthly data. We generated each of them by grouping original data by year, then applying counting and pivot aggregation that resulted in a DataFrame with year as index and column values based on hour, day of week or month. We plotted every third year for each of them as shown in figure 2. Each year followed the same patterns. The yearly crime top happened during the summer, weekly crime tops was on Fridays and daily crime tops occurred around lunch time (12:00) and in the evening. In addition to this, a day of the month analysis was performed, but it followed no pattern.

3.2 Crime Types

Most of the crimes in the dataset was represented by a fraction of the available crime types. The top 12 types, shown in table 1, represented 95.65% of the dataset. We defined a *least_frequent_columns*



(a) Monthly (b) Weekly (c) Hourly

Figure 2: Monthly, weekly and daily crime count every third year from 2001-2020

Table 1: Most frequent types of crimes.

Top 12 Primary Types	
Primary Type	Percentage
Theft	21.14
Battery	18.28
Criminal Damage	11.37
Narcotics	10.31
Assault	6.27
Other Offense	6.21
Burglary	5.66
Motor Vehicle Theft	4.60
Deceptive Practice	4.06
Robbery	3.76
Criminal Trespass	2.85
Weapons Violation	1.10
Total	95.65

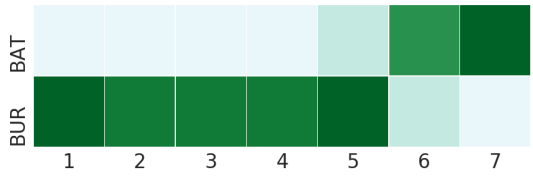


Figure 4: The weekly frequency of burglaries and batteries.

function that would take the DataFrame and a threshold as input, compute the frequency of each type, filter the ones below the threshold and then return them as a list. A user defined *renamer* function would look at each row rename those with primary types below the threshold.

Temporal analysis was performed on the renamed DataFrame. This enabled us to examine the behavior of different crimes over time. Pivot tables with time columns and type indexers was created by SparkSQL implementations. Horizontal *min-max scaling* was applied so we could compare each type with a common metric. The results were interesting and proved that there was a correlation between types and time. For instance, regular theft occurred mostly at 12 PM, while motor vehicle thefts occurred on the evening around 10 PM. Figure 4 shows which day of the week burglary and battery crimes usually happened. Many of the other types followed a similar clear distinction.

We looked at the trends for each type in order to see whether it had decreased or increased over time. Moving average smoothing over 120 days was computed for each column (type) on a generated pivot table. The final plots of each type showed that different types followed different trends. An example is shown in figure 5, where we can see that the amount burglaries has decreased over time, while weapon violation has experienced a recent increase.

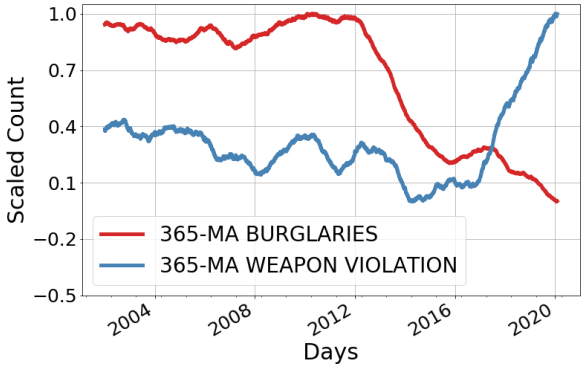


Figure 5: Scaled count of Burglaries and Weapon Violation from 2002-2020

3.3 Crime Locations

The dataset contained 180 location descriptions, such as; street, sidewalk, residence, etc. The distribution of these looked similar

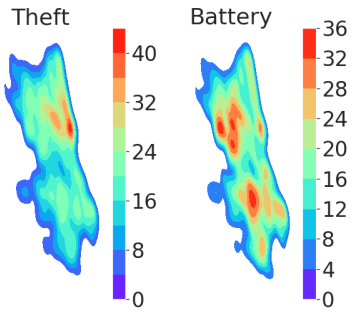


Figure 6: Crime locations of Theft and Battery

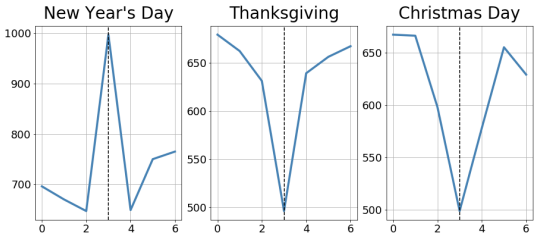


Figure 7: The effect of New Year's Day, Christmas Day and Thanksgiving.

to the primary type distribution mentioned in section 3.2. Top 15 descriptions represented 90.64% of the dataset. Temporal analysis with horizontal *min-max scaling* of a pivot table indexed by location description showed that the location descriptions also depended on time. For instance, most crimes that had happened in an alley was between 7 and 10 PM on Fridays and Saturdays, while most crime in small retail stores happened between 12 and 5 PM on all days of the week except Sundays, as they probably were closed.

We assumed that there were strong correlation between the types of crime and location description. We grouped them together and created a pivot table where primary types were indexed by location description. Vertical *min-max scaling* fixed was applied before we plotted the values. As expected, certain crime types happened at certain location descriptions. For instance, almost all Narcotics crime happened at sidewalks or streets, while most burglaries happened in apartments, residents and on the street.

Further crime location analysis involved checking whether certain types of crimes happened more frequently in special areas of the city. We filtered the range of latitude and longitudes as some data points was far away from the main areas. The remaining values were then rounded to a less specific location, so crimes with slightly different distance got grouped together. We grouped primary types together and added a count aggregation after this. KDE plots from Seaborn was used for visualization purposes. The results showed that we can cluster where certain types of crime are likely to occur. For instance, figure 6 shows that almost all theft happened in a small area of the city, while burglaries was divided by two main clusters.

3.4 Holiday effects

Holidays and special events can cause some abrupt changes on time series data that algorithms cannot capture unless they are defined. We looked at how the series behaved during the common holidays, year for year. Pythons *holidays* library[1] was used to generate U.S holidays in the state of Illinois between 2001 and 2020. User defined functions was created in order to label rows by a binary 1/0 (holiday/not holiday) and whether or not it was an observed holiday. We sorted and grouped the data by date and year and plotted each of the results. New Years Day, Christmas Day, Thanksgiving, Election day and Washingtons Birthday had clear effects year after year. In order to investigate them further, we created an algorithm that zoomed in and displayed the effects of a given holiday by detail. Figure 7 is an example how such an analysis, where we included 3 days before and after the holiday itself.

3.5 Crashes Location

We were interested to see if there were any interesting patterns in the crashes, primarily in the Police district 11 and in Chicago. Therefore we did some simple visual analysis. If there there were any intersections or street that was prone to accidents, we may be able to see this. Turns out there were some interesting intersections.

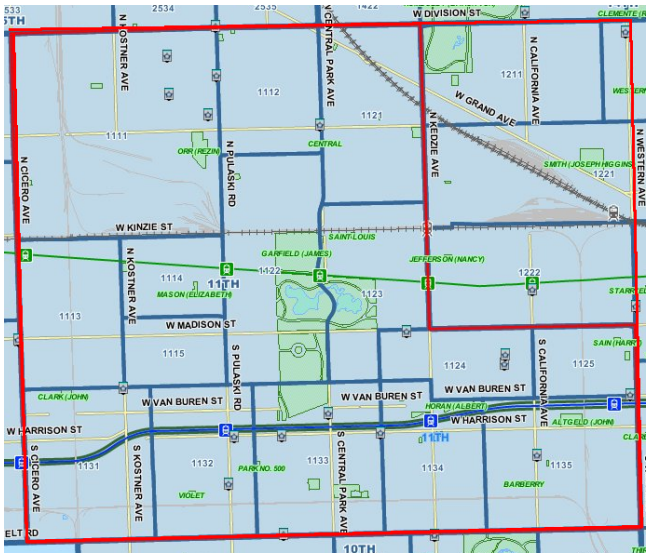


Figure 8: Picture of the 11th Police district in Chicago

[1] https://pypi.org/project/holidays/

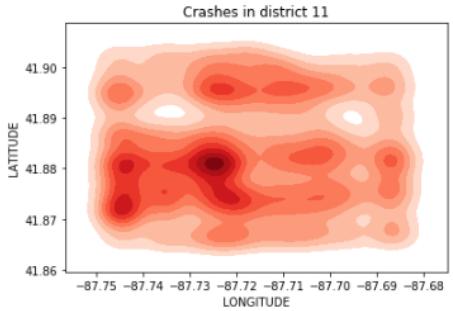


Figure 9: Heatmap plot of the crashes that has occured in district 11

From the figures above we can see some interesting finds. We can see that specifically one street, "W Kinzie ST" seems to be the most prone to traffic crashes. This made it clear as to what areas to avoid when placing the police cars. For example placing cars on each side of the road, so that it is not necessary to cross that street, and not risking getting stuck in traffic.

3.6 Anomaly Detection

What anomaly detection did was finding clear outliers. The idea was to find if there were some days that was used to fill in crimes. Filling in crimes meaning that there were some days that was used to fill in unknown crimes, or crimes that later was reported. After some analysis there were no days that looked to be days like this. Some days had amount of crimes that score above three on the test score, but it was not clear that it was data outside the pattern.

Later some anomaly detection on the location of the crimes were done instead. Location meaning the coordinates of the crime. Data that scored above three was easy to classify as faulty. When we took a close look, it turned out that the coordinates turned out to be far outside Chicago. These data points was easy to classify as faulty, and should not be in the data set.

3.7 MLlib

Features of the Chicago Crime dataset was column separated and stored in a Spark SQL DataFrame. These columns contained a variety of datatypes that had to be transformed before applying it to a machine learning algorithm by MLlib. For this purpose, we used the *Transformer* abstraction from MLlib [2], which provided useful algorithms that could transform one DataFrame into another. Transformed columns was represented by feature vectors that an *Estimator* algorithm was fit on. We chained the transformers into a preprocessing *Pipeline* which enabled us to automate the workflow. *Pipeline* stages were defined as a sequence of processing tasks that would run in order and call the transform method on the DataFrame for each run. Further implementation details is discussed in section 4.2 and 4.3.

4 CLASSIFIERS

The *CPOSS* system used a pre-trained classifier in order to determine which crimes were to occur at a given district on a given time. This section describes how we developed machine learning pipelines for such a task as well as how we tested and further improved the best suited models

4.1 Cleaning

The system we developed would have time, district and forecasted crime count per hour to base the predictions on. We dropped all features that was unavailable in a real time application from the initial DataFrame. We applied Spark SQL functions on the Date column in order to create the date derivatives; *Month, Hour, Minute, DayOfMonth, DayOfYear, DayOfWeek, WeekOfYear and Quarter*. The renamer technique that was described in section 3.2 was used with a 1% threshold, giving us a more balanced target set of 13 crime types. Crimes occurring after 2019 was filtered away since these were meant to be used for system tests in the end.

Working with 7+ million records was too much for the available cluster resources and would lead to extremely time consuming testing phases. Hence we down sampled the dataset by applying a fraction dictionary into the Spark SQL stratified sampling API. The stratified samples was without replacement and each stratum was based on the defined fractions of target labels. We re partitioned the data into $numMonths \times numYears = 228$ partitions in the last step of initial cleaning steps.

4.2 Transformers

As discussed in section 3.7, MLlib provided *Transformer* algorithms that transformed a DataFrame x into a DataFrame y. The District and target columns was the only categorical data remaining after the initial cleaning process. We encoded these with a *StringIndexer* that mapped them into a ML column of label indices in the range $[0, numLabels)$, ordered by highest frequency first.

One-Hot encoding was performed on the categorical indices we achieved after string indexing. This technique maps indexed categorical features into a binary vector containing at most a single value per row. Lets look at a record of Narcotics as an example. This crime type is the fourth most frequent crime type 1, and hence would be indexed by 4. Spark's *OneHotEncoder* would then map that to the binary vector $[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]$ given we have 13 distinct crime types.

The DataFrame contained both raw single valued features as well as generated vector columns after a base sequence of transformer processes was finished. These columns was combined into a single sparse vector by the *VectorAssembler* transformer. This format is required by multiple MLlib classifiers, such as Decision Trees and Logistic Regression models.

The feature vector we got after the *VectorAssembler* was performed could be quite high dimensional as *One-Hot encoding* was performed in an earlier process. We created a transformation step that reduced this dimension by applying *Principal Component Analysis* on the vectors. Spark provided different scaler transformers, e.g. a *StandardScaler*, that would normalize features so they would have properties of a standard normal distribution with a unit standard deviation or mean of zero. Such scaling is essential to do before applying *PCA* on the features. The reduction algorithm is based on the least squared method, which will cause too large loading on variables with large variance [12]. The *PCA* transformer would

[2] https://spark.apache.org/docs/latest/ml-pipeline.html

project the scaled feature vector into a lower dimensional space, defined by the top k components.

All of the *Transformer* algorithms mentioned above was added into a ML pipeline. Algorithms and other logic we developed from scratch was wrapped into a class that inherits from the *Transformer* class, which enabled us to add custom functionality into the pipeline as well. Custom *Transformer* classes was initiated by *inputCol* and *outpuCol*, defining which column to transform and where to add the transformed data. Each current stage object was assigned a unique ID, making it identifiable and immutable. We called *defaultCopy* on the object in order to return a new instance with any new and embedded parameter merged in. Extra safeguard was added by *input_check*, which threw an exception if the input format did not match the expected one. Finally, the *_transform* function would use SparkSQL functions to perform the desired transformation.

4.3 Pipeline testing

The machine learning pipelines contained a sequence of *PipelineStages*. The *Transformer* stages processed the data in such a way that the *Estimator* stage could fit a learning algorithm on a specified feature vector. We looked at three different learning algorithms when developing the crime type classifier; *Decision Tree, Logistic Regression* and *Multilayer Perceptron*. They were evaluated by looking at how they performed on a set of different pipelines. We developed a *PipelineTester* class, which performed, evaluated and logged all of our tests.

The first step of the testing phase was to create a *PipelineTester* object with the cleaned dataset we wanted train and test on. The class constructor would then initialize the attributes *data*, which simply was the data all algorithms would use, and *logger*, which was a dictionary with *mlp, lr* and *dt* keys. For instance, by accessing the logger with a *dt* key, all *Decision Tree* tests would be returned.

Pipeline testing could begin once the object was created. We would call custom functions for each algorithm as the next step. These functions was based on the same structure and logic, but contained model specific tasks as well. Each function took a base sequence of transformers as input. The train and test split was set to 70/30 as default, but could be modified. We could also add *normalizer, scaler* and *PCA* transformers on top of the base. The features which we wanted to transform was specified by the *fcs* input, and if we wanted to down sample the data, e.g. to balance the target labels more, we could add a fraction dictionary as well. Algorithm 1 demonstrates how the model specific functions performed tests.

A *Multilayer Perceptron* (MLP) is a type of feed-forward artificial neural network with three or more layers; an *input* layer, *output* layer and one or more *hidden* layers. The shape of the *input* and *output* layer changed for each iteration of the test, and therefore we had to use a SparkSQL map job in order to calculate the correct ones to use. We used a shallow network with one hidden layer of shape 25 as a base MLP model for testing. The base *lr* model was set with 10 as max iteration. For the base *dt* model, we used *gini* as impurity, 5 as max depth and 32 as number of bins.

We used accuracy as a metric for evaluation, where the *MulticlassClassificationEvaluator* from MLlib calculated the score for both training and testing data. The training score was included so we could see whether the model overfitted or underfitted the data.

Algorithm 1 Model specific pipeline test

```
1:  input: base, train_split, test_split, normalizer scaler, pca, fcs, fractions
2:  procedure ModelTest(input)
3:      if pca, but no scaler then return exception
4:      end if
5:      for fc ∈ fcs do
6:          for k ∈ pca do
7:              assembler ← VecAssembler(in: fc, out: Features)
8:              finalOutputCol ← Features
9:              stages ← base + assembler
10:             if normalizer/scaler then
11:                 stages ← stages + normalizer/scaler
12:                 finalOutputCol ← (normalized/scaled)Features
13:             end if
14:             if k then
15:                 pca ← PCA(in: finalOutputCol, out: pcaFeatures)
16:                 finalOutputCol ← pcaFeatures
17:                 stages ← stages + pca
18:             end if
19:             pipeline ← stages
20:             pipelineDF ← fit and transform pipeline on data
21:             (train),(test) ← randomSplit(pipelineDF)
22:             model ← Model(featuresCol: finalOutputCol, labels)
23:             Fit and transform model on train/test
24:             Logger ← accuracy, time, pipeline, fractions
25:         end for
26:     end for
27: end procedure
```

Training and testing data contained many duplicated feature inputs due to the reason that multiple crimes had occurred at the same district, on the same month, hour and day of the week. Predictions from the models returned a probability vector for each target label, which we used to calculate an additional test score named *top 3 accuracy*. If the target label was within the top 3 probabilities from the prediction, we would count that as a true prediction. Once the accuracy's where calculated for an iteration, the *logger* attribute where indexed with the current model, and stored execution time, pipeline string, fractions, and training and testing accuracy with and without top 3. Listing 1 demonstrates the use of the pipeline testing class. We create the object in line 1, and run pipeline tests for a *Decision Tree* with only the base transformers in line 2-3. Line 4-6 will run pipeline tests where additional scaling and PCA is added on the base transformers. It will test how well the *dt* model works for a list of features, with different dimensions. Listing 2 shows an example of a logger result after such a test.

Listing 1: Running pipeline tests

```
1 pt = PipelineTester(df)
2 pt.dt(base= indexers + encoders + target_indexer,\
3     fcs=fcs)
4 pt.dt(base= indexers + encoders + target_indexer,\
5     scaler = [standard_scaler], pca=[3,6,10,15,20],\
6     fcs=fcs[1:4])
```

Listing 2: Pipeline logger

```
{
    "time": 165.9,
    "pipeline": "fc_1: StringIndexer -> OneHotEncoder ->
        StringIndexer -> VectorAssembler ->
        StandardScaler -> PCA_10",
    "fractions": "No",
    "train_acc": 0.24509813845671077,
    "test_acc": 0.2461833416175531,
    "top_3_train_acc": 0.5218293534705281,
    "top_3_test_acc": 0.5257639234064712
}
```

4.4 Feature engineering

CPOSS based predictions on time and district knowledge only, which by default limited the capabilities of the classifier. As mentioned above, if the time derivatives were month, hour, and day of the week, multiple duplicated feature vectors would occur with different target labels. This would directly affect the accuracy's of the models a lot, since only one label could be predicted per unique feature vector. If 10 different types had occurred with the same date derivatives, 9 of them would not be classified no matter how perfect the model was. Adding year and minute derivatives would reduce the number of duplicates, resulting in higher accuracy on testing data. However, we excluded minute data as this seemed to be too random. The year feature made it easier for the classifier to find patterns in training and test data where an entire year had been recorded. However, if we where to classify unseen data from 2020, the 2020 feature would mean nothing and confuse the model.

We developed algorithms that added prior knowledge to each feature in order to fix the default limitation problem. This knowledge covered historical behavior of each target label. We defined four statistics to compute based on the district:

(1) Average all time, that time of the week, that hour
(2) Average past 365 days, that day of the week, (+- 1 hour)
(3) Average past 30 days, that day of the week, (+- 1 hour)
(4) Average +- 30 days last year on that day of the year, that day of the week, (+- 1 hour)

Statistics 1 captured how often a crime within a certain district had happened on the same hour and day of the week. Statistics 2-4 looked at a more recent perspective, which replaced the year feature discussed earlier. Since they contained quite few records, e.g. only 4 on stat. 3, an hour of buffer were added to each of them.

The first step of the algorithm was to add a crime count per record, meaning how many crimes occurred that day and hour, inside a certain district. We achieved this by creating a joiner DataFrame containing a group by count, which we joined with the original DataFrame. Then a tmp DataFrame containing district, crime counts and date derivatives without duplicated rows was created from the original df, reducing the overall size by ≈1.1 million rows. Next step was to group tmp by target, district and hour, and then doing an aggregation with collect_list on *DayOfWeek, DayOfYear, Hour* and *District*. We joined this with tmp by year, district and target, giving us four vectors for each row representing all similar crimes with their respective count, within the same year, e.g. all Narcotics crimes of 2018 in District 09. An example is shown below, where three similar crimes has happened within the same year. By reading the top value of each vector from left to right, we can see that 1 crime happened on a Monday day 10 (of that year) at 09:00 AM.

$$hc = \begin{pmatrix}1\\1\\3\\2\end{pmatrix} \quad dow = \begin{pmatrix}2\\4\\2\\6\end{pmatrix} \quad doy = \begin{pmatrix}10\\56\\233\\310\end{pmatrix} \quad hr = \begin{pmatrix}9\\21\\10\\17\end{pmatrix}$$

We created a user defined function; *yearly_dayofweek_hour_matrix*, which took these four vectors as input, as well as the day of week, hour and year of a record. The vectors was stacked on top of each other in a matrix format, which we then filtered until it only contained crimes on the same day of the week, +- 1 hour of the incoming record. In addition to this, a year vector of column length was added as the last row so we could filter them later on. This dense matrix format removed unnecessary data and provided a format that was easy to perform calculations on. The matrix could be explained by: row 1; day of year, row 2; day of week, row 3; hour, row 4; count, row 5; year. If we consider a record which is the top values of the vectors mentioned above of the year 2010, a dense matrix representation would look like the example below:

$$\begin{bmatrix}10 & 233\\2 & 2\\9 & 10\\1 & 3\\2010 & 2010\end{bmatrix}$$

Further, we created another user defined function; *all_time_mat* that stacked dense matrices of similar crimes for all available years. The reason we did this was because we had to access all historical data when performing statistics 1. Max crime length was defined by the year that had most crimes of this type. Years with less crimes was then padded with the remaining length so the shape of the stacked matrix was in correct format.

With *stacked_padded_matrices* available, all statistics could be calculated by the user defined functions; *all_time_avg*, *avg_past_year*, *avg_past_month* and *avg_last_year_months*. They filtered a stacked and padded matrix by year and day of the year, including only records that had happened before itself and within the preferred range of days. The total count was then divided by number of weeks, producing an average number to be returned.

Districts, statistics and date derivatives collected by SparkSQL was used to generate a dictionary by the *spark_generate_stats* function. Another function named *get_probability_vectors* would then index the dictionary by collected lists, and generate a combined feature vector of length $4 \times 13 = 52$, containing all available statistics for each target type.

We used pandas to generate hourly data for each district and crime type between 2020.01.01 and 2020.01.31, resulting in 15862 rows. Dates after the first week would not have 30 recent days to look at since the week(s) before were generated. Therefore, minor changes in the statistics functions had to be done when dealing with such future generated data. Statistics 2 was changed to past 365−dayofyear days, and statistics 3 was changed to past 60−dayofyear days. We fetched the most recent records of each crime type within a certain districts at a given hour and day of the week. This DataFrame, together with the future DataFrame,

was put into a *get_future_probabilities* function that mapped future data with most similar recent crime and calculated the statistics. In a similar way as with the historical data, *get_probability_vectors* parsed a generated future statistics dictionary and returned feature vectors for each possible happening in January 2020. We zipped all necessary data into a Spark DataFrame that could be used by classifiers that was trained on historical statistical data.

4.5 Parameter tuning

MLlib provided *ParamGridBuilder*. which enabled us to construct a grid of parameters suited for a model. The *CrossValidator* functionality would then split data in k folds which then were to be used as different training and testing datasets. An average of each model score would be computed and returned at the end. *ParamGridBuilder*, pipelines and an evaluator would be wrapped into the *CrossValidator* instance and fitted on the entire dataset.

The functionality provided by MLlib was easy to use. However, it was hard to gain additional control on the testing part. For instance, logging custom results, e.g. top 3 accuracy, would require a lot of changes. The main classifier we focused on was the Decision Tree, which do not have many parameters to tune in MLlib. Hence, we created custom tuning functions that looked at depths and impurity of the tree, as well as different PCA approaches. Custom performance metrics, such as execution time and top 3 test error was implemented into the search, and logged into a JSON file.

5 FORECASTERS

5.1 Prophet

Prophet is a procedure for forecasting time series data, and is developed by Facebook [4]. It is a tool that works best if the data has strong seasonal changes. This section describes how we used this library for two forecasting tasks; Crime and traffic frequency.

5.2 Forecasting Crimes in Parallel

Popular libraries such as *numpy*, *sklearn* and *pandas* could be applied on the Spark DataFrame using *pandas-udf*. We looked at the *GROUPED_MAP* attribute, which made it possible for us to forecast multiple time series using prophet, at the same time. This was useful, as we had multiple clusters available that could process district series in parallel, and provide a processed DataFrame that contained all the results without further implementation.

As discussed in section 3, trends and holidays affects time series models. We added additional UDFs that fetched and joined holiday labels into the test and training data. Crimes after 2020 was filtered out for training data and used for testing purposes. We grouped the filtered datasets by (Day/Hour/Month etc.), District and holiday before applying an aggregated count. District was chosen as a location attribute since they contained sufficient amount of data to develop a forecast. Dealing with lower levels, such as Beats, resulted time series with way to low variance. We added cutoff points for the training data before joining it with test data, which we then could pass to the prophet algorithms.

Prophet expected a *pandas* DataFrame with a certain format. Input data was queried by the specified cutoff point, so it could divide train and test sets internally for each run. Additional filtering involved renaming columns to the expected format and sorting it based on dates. The prophet models was set with custom seasonality attributes, based on the data exploration we had gone through earlier. The test data was casted to a *pandas* DataFrame, before the defined model could predict values, join them on date, and return a DataFrame containing a predefined Schema.

5.3 Forecasting traffic

Our goal was to predict where the traffic was going to the slowest, so that the police car could avoid these areas. Unfortunately some challenges occurred, but that is discussed in a later chapter9. The most reasonable approach was predicting daily traffic, for a smaller area. Then we could at least check if it was a busy are or not, in general.

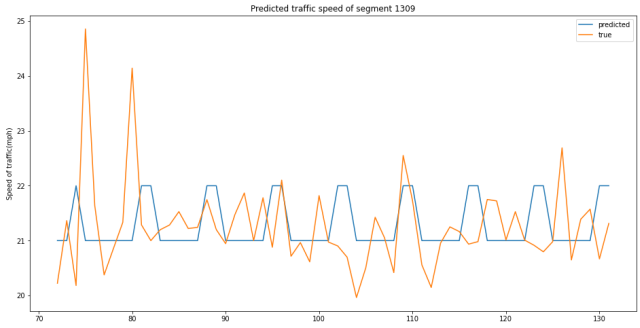


Figure 10: Traffic forecasting for segment 1309

Segment 1309 is a road in Chicago, and the one with the most records in the data set. The data the forecasting model was trained on, were divided into days. This was because it was not enough data so that we could divide it into hours. The way it was calculated was that we took the average of every traffic tracker row that was registered on the same day. As one can see in the figure 10 the speed of the vehicles only ranges from about 21 to 22 mph on our predictions. Because we took the mean of every row on each day, the data has not much variation at all. Even the true data only varies from 20 to about 24, when calculated this way. This is one of the worst examples we have of the forecasting we did, but it shows that the data sets are not very suitable for this method of forecasting. As we can see it has only learnt a pattern from previous data, and repeats this.

When forecasting the amount of traffic crashes we get somewhat the same results. It will follow a strict pattern. If we predict traffic crashes hourly, the pattern will not be as strict, and will have more of a trend that it follows. But in general it is nowhere near the true line.

6 EXPERIMENTAL EVALUATION AND RESULTS

We present multiple conducted experiments with their results in this section, as well as detailed description of the system they were run on.

6.1 Experimental setup

The cluster setup consisted 4 VMs. One acted as a master node, and the other three acted as slave nodes. Specifications of the VMs are listed below. The three slave nodes had 40 GB of disk space each. These were used to store the data. The slaves had about 2.9 GB of RAM, from what we could tell.

- Ubuntu 16.04.6 LTS
- Hadoop 3.1.3
- Spark 3.0.0
- Python 3.7.6
- Anaconda environment on each slave
- Jupyter notebook

6.2 Hadoop vs Spark

We wanted to do some comparisons between Hadoop MapReduce and Spark, specifically in run times. This was to see how fast the scripts were running in the different enviroments. The exact same MRJob script is running on both "Local" and "Hadoop". "Local" was run with the file on the local VMs, and "Hadoop" was run with the file on the HDFS. From table 2 it is clear that the difference between locally running the MRJob and running it on the Hadoop cluster is minimal. There are a few reasons as to why the differences are that small. Mainly, it is because it is quite a small MRJob. It looks like the overhead to start a MRJob on the cluster, is bigger than the benefits from running it on the cluster.

However, when running the spark script, it is almost halved. The reasons as to why Spark is faster, at least at this smaller task, is because it prevents unnecessary I/O with the disks. As a big portion of this small task on MapReduce contains making temporary directories and copying files, Spark is quicker because it does not have to do these things.

Table 2: Comparing a mrjob running locally, on the Hadoop cluster and on Spark

Local	Hadoop	Spark
04:56	04:45	02:19

It was also interesting to experiment with how the Hadoop cluster handled copying files back and forth, and specifically the file that the MRJob was running on. As table 3 shows, there are big differences on where the file is. "File locally" meaning that the MRJob was run on the cluster, and using the local file. "HDFS" meaning that the MRJob was running on the cluster, and using the file being put on the Hadoop file system. This suggests that Hadoop uses more or less a minute to copy the file from the local VM to the Hadoop cluster, and so it is beneficial to put the files on HDFS.

Table 3: File locally or file on HDFS.

Local vs HDFS	
Local	HDFS
05:47	04:45

6.3 Crime Type Classifier

We ran pipeline tests on a small subset of the original dataset (≈150 000 rows). We evaluated six different feature columns, namely:

(1) All time derivatives
(2) No yearly perspective (Hour,Month,DayOfWeek)
(3) Some yearly perspective (Year,Hour,Month,DayOfWeek)
(4) No seasonal perspective (Year,Hour,DayOfWeek)
(5) Precise daily perspective (Hour, Minute)
(6) Imprecise daily perspective (Hour)

with different transformer pipelines containing bases with and without scalers and PCA. The district feature was part of all tests. Totally 92 tests were conducted over 5.3 hours.

Table 4: Pipeline testing results

Model	acc.	top 3 acc.	ex.time
mlp base (fc_5)	0.27	0.55	220
mlp base (fc_3) + pca_20	0.25	0.54	297
lr base (fc_5)	0.26	0.54	135
lr base (fc_1) + pca_10	0.24	0.53	175
dt base (fc_5)	0.27	0.54	139
dt base (fc_3) + pca_15	0.24	0.52	254

Top base and PCA performance for each model is listed in 4. All models scored best on the fc_5 (Hour, Minute). We think this is because the models struggled to distinguish data with maximum hourly details. The *lr* model performed worse with PCA and improved as k increased. PCA did not improve *mlp* or *dt* either. These tests made us realise that a system purely based on date and district knowledge had to add additional data in order to fix duplicated feature values, as discusses in section 4.4. These basic models could not be used at all in a real time situations, so we performed feature engineering on (≈500 000 rows) and tuned a Decision Tree model, which seemed to suit our case well based on the pipeline tests. The final test error rates are shown in table 5.

Table 5: Decision Tree Classifier

Depth	test error.	top 3 test error.	ex.time
5	0.303	0.203	**28**
10	0.092	0.014	30
20	0.058	0.009	40
25	**0.058**	**0.008**	57
30	0.06	0.008	88

Feature engineering improved the model drastically, where the top model achieved an accuracy of 0.94 compared to the best model before, achieving 0.27. Further by including the 3 most likely crimes, we achieved an accuracy of 0.992. We trained on crimes earlier than 2015 (≈400 000 rows) and tested on data from 2016-2019 (≈75 000 rows) with a set of different depths. The model seemed to improve as the tree got deeper, until a point around 25. All testing data was at least 1 year later than all training data, and all features was engineered only including previous knowledge. Such a satisfying result gave us confidence that the model would perform well on future generated data that CPOSS would use.

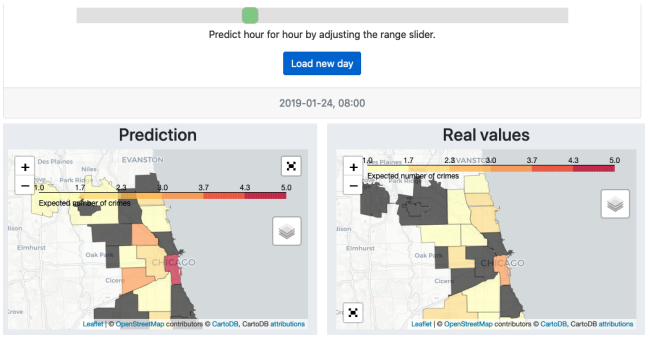


Figure 11: Simulation of future generated data on the left, true outcome on the right

6.4 Forecasting Crime Frequencies

We implemented a parallel way of forecasting crime frequencies across the city. This worked fine, and did probably speed up the entire process. We looked at different time intervals during the testing phase, such as; 10 min, 30 min, 1 hour, 1 day, 1 week and 1 month. Tests was performed both on district and Beat level.

Prophet did not work well whenever we tried to predict crimes on Beat level, below daily time intervals. Most of the Beats had none crimes per data point, which algorithms struggled to fit on. District level data contained more crimes, and was then a better pick even tho they provided a less accurate location description. Predicted data varied a lot in performance. Algorithms from Prophet did not capture randomness well, and followed way to strict trends. It would be interesting to see whether a classifier or regressor could perform better, by applying additional information through feature engineering. However, we focused purely on forecasting algorithms only for this task in this paper.

7 CPOSS SIMULATION

We developed a *flask* server that demonstrated basic usage of CPOSS. The page contained a date picker for January 2020, as well as an hour slider. These time settings could be used to generate predictions for each district, that would be displayed on a dynamic *folium* map. Such a map made it easy for the user to navigate inside the city and explore districts that had color codes based on the forecasted crime frequency. GeoJSON borders separated the districts and implemented individual statistics that would pop up if the user clicked on it. HTML code containing the probable crime types at that time and district was generated on a user click, and presented in a popup box.

The true outcome of each filter was presented next to the predicted data. This made it easy for us to evaluate how well the overall system worked. The simulated demo server was just a simple prototype, and could be extended by many means. For instance, the forecasted traffic data could be integrated, as well as other beneficial statistics for the police.

8 CONCLUSION

Using the methods we did to forecast the time series data had did not work as hoped. The data turned out to have too little variation, at least at the scale we wanted to forecast it as. Prophet, the tool we used to forecast the data, works best if the data has big seasonal variations. This was the biggest challenge in terms of determining the route of the police car.

This project [2] was mainly done using Spark. Spark allowed us to use familiar tools like jupyter notebook and to use data frames. It would have been a hassle if we tried to do the same things we did in a MRJob script for example, as this was a typical Data Science task.

As for CPOSS and recommending a place to put the police cars, it would be to avoid streets with low speed. So that the chance of getting stuck in traffic is low. CPOSS works great in terms of predicting what type of crime it will be.

9 CHALLENGES

There were some limitations and challenges in this project. Partly because of the data sets, and partly because of our use case.

9.1 Prophet/Forecasting

We noticed that a big downside with forecasting our data is that it has not big enough changes. It can vary of course vary, but in general it does not have big seasonal changes. Prophet is a tool that works best if the data have strong seasonal effects. If we want to predict traffic flow in Chicago, we know that there are times that are busier than others, e.g. rush hour. Rush hour will happen every weekday at around 08:00, and around 17:00. This means that the data does not have great seasonal changes, and this rush hour will happen almost every weekday of the year. It might have worked better if we predicted on a bigger scale, such as weekly traffic flow, but then this would not help us much in our problem. Knowing what week that will have more traffic jams does not help when calculating where to place a police car to avoid traffic at a certain time of day.

9.2 Limitations of data

This problem were significant when we were trying to forecast the traffic in Chicago. The data set consists of 19.6 million rows, and it still was a limitation. There were two reasons as to why this was a challenge. Firstly, when we divided this data set into smaller ares, such like a street, there were limited data. Not enough data for each street was recorded to provide enough data to forecast in a reasonable time frame, e.g. forecast for every hour of the day. We could forecast the traffic for each day, but that was not very helpful regarding our end goal. Secondly, when we did not divide the data in to smaller areas, it did not make sense to forecast for the entire Chicago. Knowing that the traffic is predicted to be high at 19:00 in the entire Chicago, does not really help to know which specific street is going to be busy at 19:00 when placing a police car.

9.3 Challenge predicting locations

Our biggest challenge was predicting coordinates. We could not get this done. The goal was to predict where the next crime would happen. We also wanted to predict where the next crash was going to happen, and where the traffic jam was going to be. We ended up predicting the crime in smaller areas of Chicago, which could be sufficient. The problem occurs if the road is closed because of a

crash, and the police car needs to take a longer route. There may be some areas that could have multiple high traffic roads that might be difficult to cross. Therefore it is hard to decide where the police cars should be placed. One workaround could be to have models to forecast for smaller areas to get more specific locations, but then we have the problem with limited data on each place.

9.4 Limitation of resources

Especially the amount of RAM on the slave nodes caused us trouble. We continuously got error messages when training the classifiers. This was due to the reason that the training data was too big for the Spark configuration on the cluster. We tested many configurations, but the available resources was never able to handle the entire dataset before throwing memory exceptions.

10 FURTHER WORK

Further work would be to find some way to more precisely predict the location of the events that we have discussed, and especially traffic jams and traffic crashes. This makes it much easier to determine where the police cars should be placed, and what routes they should take to the destination of the next crime. This is mainly limited by data, at least for forecasting traffic. There are other traffic related data sets that is provided by the City of Chicago and it may be a possibility that we can combine some of these data to provide sufficient data and thus predict more precisely. It can also be worth trying other methods for forecasting, instead of Prophet. A method that is less dependent on seasonal varitaion of the data could improve the predictions quite a lot.

REFERENCES

[1] Chicago Police Department. 2020. Chicago Traffic Tracker Data set. https://data.cityofchicago.org/Public-Safety/Crimes-2001-to-present/ijzp-q8t2

[2] Bjarte Botnevik Eirik Sakariassen. 2020. Chicago Police Officer Scheduling System. https://github.com/Eiriksak/CPOSS

[3] Uber Engineering. 2020. Time Series Forecasting in Uber. https://eng.uber.com/forecasting-introduction/

[4] Facebook. 2020. Homepage of Prophet developed by Facebook. https://facebook.github.io/prophet/

[5] Zaisha Zamal Khan Mohammad Habibullah Amit Kumar Das Jesia Quader Yuki, Md.Mahfil Quader Sakib. 2019. Predicting Crime Using Time and Location Data. https://www.researchgate.net/publication/335854157_Predicting_Crime_Using_Time_and_Location_Data

[6] Nikolay Laptev. 2020. Anomaly detection in Uber. https://forecasters.org/wp-content/uploads/gravity_forms/7-c6dd08fee7f0065037affb5b74fec20a/2017/07/Laptev_Nikolay_ISF2017.pdf

[7] Zack Lim. 2020. Classification with ML: Predict Crime Type. https://www.kaggle.com/heng8835/classification-with-ml-predict-crime-type

[8] City of Chicago. 2020. Chicago Traffic Crashes. https://data.cityofchicago.org/Transportation/Traffic-Crashes-Crashes/85ca-t3if

[9] City of Chicago. 2020. Chicago Traffic Tracker Data set. https://data.cityofchicago.org/Transportation/Chicago-Traffic-Tracker-Historical-Congestion-Esti/77hq-huss

[10] Sean J Taylor and Benjamin Letham. 2018. Forecasting at scale. *The American Statistician* 72, 1 (2018), 37–45.

[11] Eidell Wasserman and Carroll Ann Ellis. 2007. Impact of crime on victims. *A. Liew (éd.), Participant text: National victim assistance academy. Site visité à: http://www.ccvs. state. vt. us/joomla/index. php* (2007).

[12] Svante Wold, Kim Esbensen, and Paul Geladi. 1987. Principal component analysis. *Chemometrics and intelligent laboratory systems* 2, 1-3 (1987), 37–52.