UNIVERSITY OF PISA

COMPUTER SCIENCE DEPARTMENT

Data Mining - Master Degree in Computer Science

Data Mining Project Report

Authors:

Angelo Nardone (548522) - a.nardone5@studenti.unipi.it
Lorenzo Spiridioni (619457) - l.spiridioni.unipi.it

ACADEMIC YEAR 2023/2024

Contents

Abstract

The following report outlines all the crucial steps addressed in the Data Mining project: from data understanding to time series analysis. Our work will specifically follow the project outline, which can be observed at the following link. The project was entirely developed in Python, using Jupyter as the working environment. You can find our code on the following GitHub page: DataMining. The main objective of this paper is to provide a clear exposition of each step performed, motivating the choices made and illustrating the results obtained. The continuation of the report will then contain in-depth commentary on the achieved results, the imported libraries, and the code choices used to support the data mining process.

1 Data Understanding and Preparation

Throughout this report, we will be working with three datasets provided in CSV format: *incident.csv*, *povertyByStateYear.csv*, and *year_state_district_house.csv*. In this initial section, we will delve into the exploration of these datasets, optimizing them, and, upon the conclusion of our work, combining them into a unified dataset.

1.1 Data Understanding

As mentioned earlier, the data understanding task revolves around analyzing the datasets and integrating them by addressing duplicated values, rectifying missing values, and resolving potential outliers.

1.1.1 Data Semantics

Before manipulating the data, the first task performed is to understand the semantics of each attribute we need to work with. For each attribute we also differentiate between the original data type and the cast data type.

Feature name	Initial Type	Cast Type	Description
`year`	`int64`	`int64`	Year in which the data is taken.
`state`	`object`	`object`	State where the vote takes place.
`congressional_district`	`int64`	`int64`	Congressional district where vote takes place.
`party`	`object`	`object`	Winning party for corresponding congressional district in state.
`candidatevotes`	`int64`	`int64`	Number of votes obtained by winning party in election.
`totalvotes`	`int64`	`int64`	Number total votes for corresponding election.

Table 1: Features overview of the District DataFrame.

Feature name	Initial Type	Cast Type	Description
`state`	`object`	`object`	State where the vote takes place.
`year`	`int64`	`int64`	Year in which the data is taken.
`povertyPercentage`	`float64`	`float64`	Poverty percentage for the corresponding state and year.

Table 2: Features overview of the Poverty DataFrame.

Feature name	Initial Type	Cast Type	Description
date	object	datetime	Date of incident occurrence.
state	object	object	State incident took place.
city_or_county	object	object	City incident took place.
address	object	object	Address incident took place.
latitude	float64	float64	Latitude of the incident.
longitude	float64	float64	Longitude of the incident.
congressional_district	float64	int64	Congressional district where incident took place.
state_house_district	float64	int64	State house district where incident took place.
state_senate_district	float64	int64	State senate district where incident took place.
participant_age1	float64	int64	Age of one (randomly chosen) participant.
participant_age_group1	object	object	Age group of one (randomly chosen) participant.
participant_gender1	object	object	Gender of one (randomly chosen) participant.
min_age_participants	object	int64	Minimum age of participants.
avg_age_participants	object	float64	Average age of participants.
max_age_participants	object	int64	Maximum age of participants.
n_participants_child	object	int64	Number of child participants.
n_participant_teen	object	int64	Number of teen participants.
n_participant_adult	object	int64	Number of adult participants.
n_males	float64	int64	Number of males participants.
n_females	float64	int64	Number of females participants.
n_killed	int64	int64	Number of people killed.
n_injured	int64	int64	Number of people injured.
n_arrested	float64	int64	Number of people arrested.
n_unharmed	float64	int64	Number of people unharmed.
n_participants	float64	int64	Number of participants.
notes	object	object	Additional notes about the incident.
incident_characteristics1	object	object	Incident characteristics.
incident_characteristics2	object	object	Additional incident characteristics.

Table 3: Features overview of the Incidents DataFrame.

In Table 1 we find information about the dataframe `district`, in Table 3 we find information about the dataframe `incidents`, and in Table 2 we find information about the dataframe `poverty`.

1.1.2 Utility Function

Before we begin manipulating the data to enhance its quality, we have created a section where we have defined some **utility functions** that we have used throughout the data understanding and preparation tasks. We categorize the utility functions into three types:

- **Plot functions**: for plotting various types of graphs: box plots, bar charts, scatter plots, pie plots, and histograms.
- **Fill functions**: for filling NaN values in specific columns with the mean or median of non-NaN values in the same columns.
- **Correlation functions**: for intelligently extracting correlation results.

1.1.3 Type Casting

In all three datasets, an attribute casting operation was necessary due to incorrect or generic initial data types, aiming to obtain the data format described in Tables 1, 3, and 2. Specifically, we convert the values of columns representing dates to the `datetime` type of pandas using `to_datetime()` command, and the values of columns with numerical information to `float64` and `int64` using `to_numeric()` command. During this operation, values that did not conform to their designated type (e.g., a string or an infinite value cast to an integer) were set to null. To accommodate null values, we opted for Pandas nullable data types (e.g., `float64` instead of `np.float64`).

We also noticed that some columns, which were supposed to contain `int` values, instead had `float` values (such as `min_age_participants`). Therefore, we proceeded to approximate these decimal values to the nearest integer using the `round()` command. Additionally, we removed **unnecessary whitespaces** (leading and trailing whitespaces) from string columns of type `object` using the `strip()` command.

1.1.4 Dropping Duplicates

During the initial exploration of our data, one crucial step was to examine and manage potential **duplicate rows**. Duplicate entries have the potential to distort our analyses and produce inaccurate results. To address this concern, a search was conducted across the datasets to identify and examine duplicate rows. The final result is that in the `incidents` dataframe we found 253 duplicate rows, while in the `poverty` and `district` dataframes no duplicate rows were identified. Following this identification process, we proceeded to eliminate all duplicate rows using the `drop_duplicates()` command. This ensures that each record is retained only once within the dataset.

1.1.5 Dropping Useless Attributes

At this point, we proceeded to analyze the various columns individually. First, we observed that several columns in the `incidents` dataframe were not relevant to our objectives. Therefore, we decided to **remove them from the dataframe** using the `drop()` command. In particular, we eliminated 8 columns, namely:

- `participant_age1, participant_age_group1,` and `participant_gender1`: These three columns all had randomly inserted data. We do not consider this data to be significant.
- `address`: We removed this column because the data within it was scattered, as revealed by the `describe()` command. Additionally, we deemed the data to be of little interest for our future analysis.
- `state_house_district` and `state_senate_district`: We eliminated these columns because contained scattered, incomplete, and inaccurate data.
- `notes`: We removed this column because we believed it could not be generalized to extract useful information. Out of the 150,000 non-null entries in this column, 135,000 had only one occurrence, indicating sparse values.
- `incident_characteristics2`: We eliminated this column due to its numerous null values. Moreover, it provided redundant information already present in part in the `incident_characteristics1` column.

1.1.6 Manage Outliers and Noise Detections

In this section, we address the **management of outliers** during the analysis of the columns. We explored the distributions of attributes in all three dataframes using various methods.

For attributes describing a date, we plotted **histograms, pie charts, and bar charts** to study the distribution and check for the presence of outliers. Note that we replaced the `date` column in `incidents` with four new columns: `year`, `month`, `week`, and `day`. We identified noise only in the `year` column of `incidents`, where the years 2028, 2029, and 2030 appeared. We chose not to eliminate these occurrences but to replace them with 2018, 2019, and 2020, respectively (more detailed explanations are available in the notebook).

For categorical and string-type attributes, we directly assessed correctness by printing their values and distribution. In this case, we found outliers in the `party` column of the `district` dataframe, which we set to null.

Finally, for numerical attributes, we primarily used **box plots and histograms** to identify possible outliers. Almost all numeric columns had outliers, which we decided to set as null. In some columns we performed also **other specific checks**. Some checks were related to the well-definition of data, such as ensuring that `povertyPercentage` was a number between 0 and 100. Other checks related to data consistency, for example, ensuring that `_killed` was not greater than `n_participants` (and all columns with specific participant information) or that `n_males` + `n_females` was not greater than `n_participants`. Again, in cases where these conditions were not met, we set the values of specific columns to null.

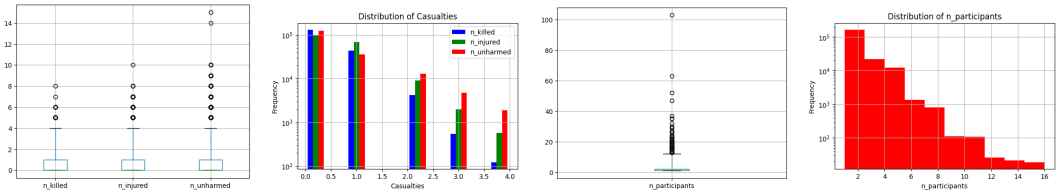


Figure 1: An example of how we found outliers in multiple features.

1.1.7 Manage Null Values

For all three dataframes, we decided to handle null values in the same way. Recall that null values also included outliers that we had transformed into null. In general, our approach was as follows:

- If null values in any column (numeric or non-numeric) did not exceed 10% of the total size of the dataframe, we chose to **delete the rows where they were present** using the `drop()` command. This approach stems from the fact that when dealing with a large amount of data: so rather than keeping a small number of rows with dirty values, we deemed it appropriate to eliminate them and retain only the clean rows.
- If null values in any column exceeded 10% of the total size of the dataframe, we replaced these values with the **median** of the column using `fill_median()` function defined in Section 1.1.2. Note that the columns where this occurred are all of numeric type, and null values never exceeded 35% of the column.

1.2 Data Preparation

In this section, we focus on describing how we built our new dataframe by merging the three previously described dataframes. The resulting dataset is cleaned and enriched with new features derived from the original ones. Finally, we proceed to print the correlation matrix.

1.2.1 Adding New Indicators

At this point, having obtained clean dataframes with no outliers and no null values, we move on to define new indicators, present in Table 4. After adding these new indicators, we then

move on to eliminate redundant columns in our dataframes. Specifically we eliminate 10 columns which are: `min_age_participants`, `avg_age_participants`, `max_participants`, `n_males`, `n_females`, `n_particip_child`, `n_participants_teen`, `n_participants_adult`, `candidatevotes` and `totalvotes`.

Feature name	Dataframe	Type	Description
`incidents_state_year`	`incidents`	`int64`	Incidents for each year in each state.
`incidents_city_year`	`incidents`	`int64`	Incidents for each year in each city.
`age_range_indicator`	`incidents`	`int64`	Difference between the max and the min age of participants.
`age_combined_index`	`incidents`	`float64`	Combination of min, avg and max age of participants.
`percentage_males`	`incidents`	`float64`	Percentage of males over the number of participants.
`percentage_adults`	`incidents`	`float64`	Percentage of adults over the number of participants.
`severity_index`	`incidents`	`float64`	Killed+injured over participants.
`poverty_difference`	`incidents`	`float64`	Difference between state's and United States poverty percentage in the same year.
`poverty_comparison`	`incidents`	`float64`	Categorization based on poverty difference index.
`percentage_votes`	`incidents`	`float64`	Percentage of votes of winning party candidate.

Table 4: New features overview.

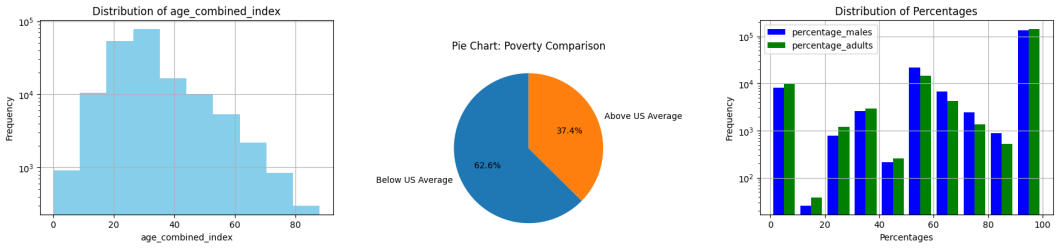


Figure 2: Some distributions of the new indicators.

1.2.2 External Indicators

We also take from two external datasets new features to add to our `incidents` dataframe. Since these new features however when added to our original dataframe have many null rows, we decided to add them to a new dataframe called `external_incidents`. From this, we then remove the null rows, so that we get a dataframe smaller in the number of rows than `incidents`, but with more columns. We will use this dataframe for analysis in the clustering part. The new features added are as follows.

Feature name	Type	Description
`population`	`int64`	Number of the city's population.
`Median Age`	`float64`	Average age of the inhabitants of each city.
`Number of Veterans`	`float64`	Veterans present in each city.
`Foreign-born`	`float64`	Non-citizen inhabitants of the United States present in each city.
`Average Household Size`	`float64`	Average size of families for each city.

Table 5: External features overview of the external_incidents DataFrame.

Finally, using these new columns, we create a new index called `welfare_index` which is

calculated as:

$$\frac{0.4 \cdot \texttt{Median Age} + 0.1 \cdot \texttt{Number of Veterans} + 0.2 \cdot \texttt{Foreign-born} + 0.3 \cdot \texttt{Average Household Size}}{4}$$

1.2.3 Join Dataframes

In this part, the integration process involves combining the three dataframes along their common columns using the `pd.merge()` command. Initially, we merge the `incidents` and `poverty` dataframes using the shared columns `year` and `state`. Subsequently, this newly formed dataframe is merged with the `district` dataframe using the common columns `year`, `state`, and `congressional_district`. Throughout these merging operations, the columns of the `incidents` dataframe serve as the discriminating factors. This approach ensures that the resulting dataframe, called `join_dataframe`, retains several rows equal to that of the `incidents` dataframe. By aligning the join operations with the `incidents` dataframe, we maintain consistency and coherence with the primary dataset of interest. Finally, we do the same join operation but using the dataframe `external_incidents`. We call the newly created dataframe `join_external_dataframe`.

1.2.4 Correlation

Finally, after having joined the dataframes, we conclude the data preparation phase by printing the **correlation matrix**. In Figure 3 below, the results of this process are presented, which is carried out exclusively on the columns of the `join_dataframe` containing numerical values.

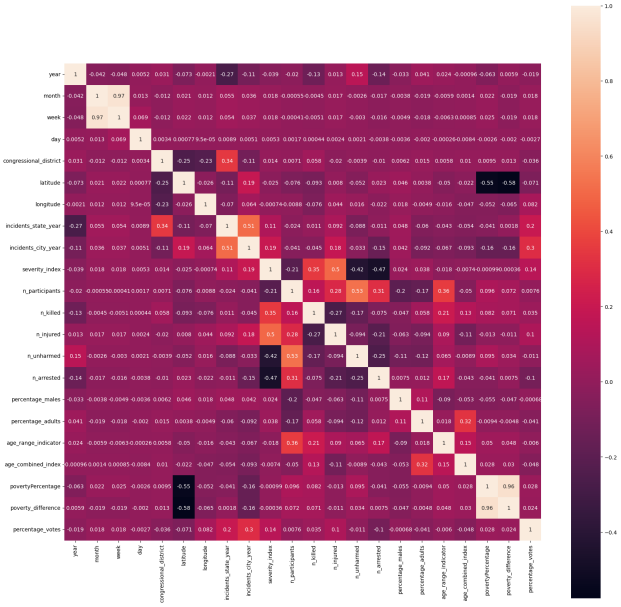


Figure 3: Correlation Matrix.

Moreover, we use two functions defined in Section 1.1.2: one is called `get_redundant_pairs()` and the other is called `get_top_abs_correlation()`. We use these functions to print the top 20 pairs with the highest absolute correlation index. At the end of the process, we observe that the pair with the highest correlation index is (`latitude-povertyPercentage`), with an index of 0.55.

2 Clustering Analysis

In this section, we delve into clustering, a data analysis technique that groups similar objects into distinct clusters. We will explore four clustering methods to uncover patterns and relationships in our data.

2.1 Preprocessing

In the preprocessing phase, we **import the datasets** created at the end of first section of our analysis. To prepare the data for clustering, we categorize features into categorical and numerical, allowing for distinct treatment. Next, we carefully select features, ensuring that **highly correlated columns are excluded** to avoid redundancy (such as `n_killed` and `severity_score`).

To gain an initial visual insight into the data distribution, we generate a pairplot with the command `sns.pairplot()` from the Seaborn library. This graphical representation aids in identifying potential patterns and relationships among features.

Further, we explore two **normalization techniques**: in particular `StandardScaler()` and `MinMaxScaler()`. After careful consideration, we opt for MinMaxScaler due to its better compatibility and effectiveness in our clustering context, compared to StandardScaler.

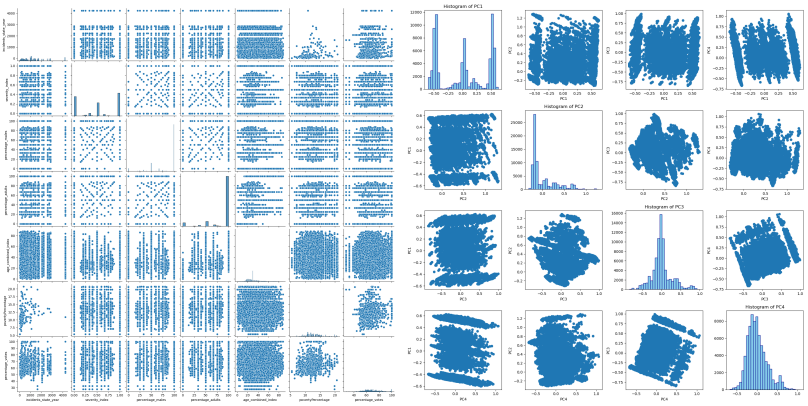


Figure 4: Pairplot generated for the datasets normalized with MinMax Scaler and with PCA.

2.1.1 PCA

In the preprocessing phase, **Principal Component Analysis (PCA)** emerges as a powerful technique for handling high-dimensional data that lacks clear visualizations or distinct concentrations. PCA aims to reduce dimensionality while preserving vital data information. The method involves analyzing covariance between feature pairs, identifying how variations in one variable correspond to variations in another. These results are encapsulated in a covariance matrix. PCA then seeks linear combinations of features that best explain this matrix, determining the linear models that capture maximum variance. Eigenvectors and eigenvalues, representing directions and coefficients, form the **Principal Components**. The choice of dimensions hinges on the explained variance of each feature. The selected principal components reorient the data, effectively reducing dimensions.

Opting for the minmax scaler due to superior results, we incorporate it into our preprocessing, creating a new block for clustering analysis normalized with PCA. PCA reveals that selecting **just four features** captures the key information in our dataframe. A pair-

plot, generated post-normalization with PCA, visually explores these components, providing further insights into data patterns.

2.2 Methods

2.2.1 K-Means

In the exploration of **K-means** clustering, we delve into the impact of different normalization techniques, including MinMax Scaler, Standard Scaler, and PCA. Notably, the results from Standard Scaler normalization prove ineffective, prompting us to exclude this method from further analysis. To determine the optimal number of clusters k, we employ the **Elbow Method** based on the **Sum of Squared Errors (SSE)** and **Silhouette Score**. After iterating through potential k values, we identify that $k = 4$ exhibits the most significant change in SSE, indicating an optimal number of clusters.

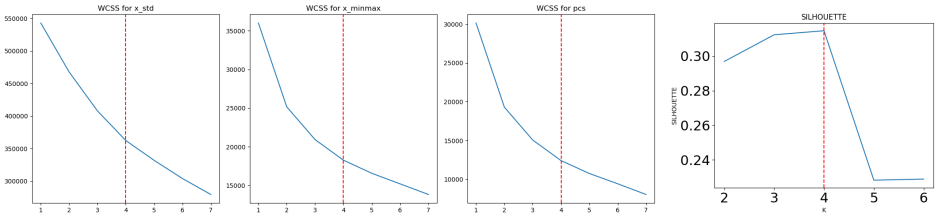


Figure 5: Elbow method using SSE for all normalization methods and using silhouette for MinMax.

Subsequently, we execute the K-means algorithm with $k = 4$ on the dataframes normalized with MinMax Scaler and PCA. We analyze the resulting clusters, inspecting their composition, sizes, and centroid variations.

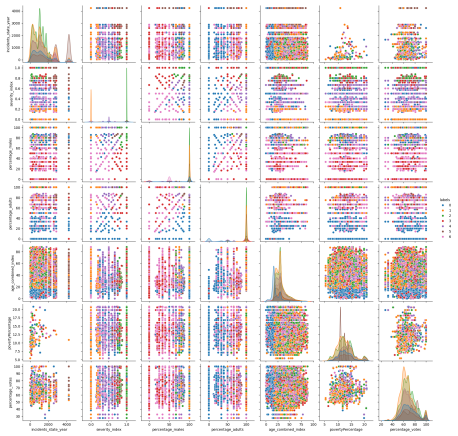


Figure 6: Pairplot generated for K-means clustering, with distinct clusters differentiated by color, normalized using MinMaxScaler.

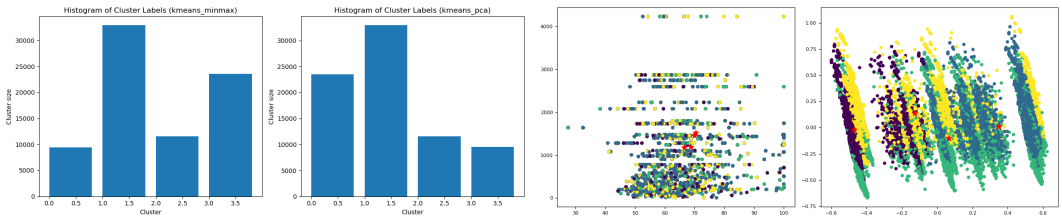


Figure 7: Cluster sizes and representation in two dimensions with centers for MinMax and PCA.

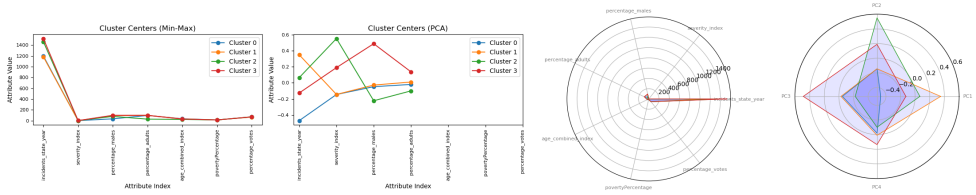


Figure 8: How the centers vary during the clustering algorithm for MinMax and PCA.

Further insights are gained through the evaluation of **Silhouette Score, Separation**, and **SSE**, providing comprehensive metrics for cluster quality, inter-cluster separation, and overall clustering performance. These results are presented in Table 6.

Metric	MinMax Scaler	PCA
SSE	18263.98	12412.99
Separation	1.3119	1.0974
Silhouette	0.3145	0.4140

Table 6: K-means Clustering Metrics for $k = 4$

After we have completed clustering based on our numerical attributes, we have incorporated the categorical attributes and observed their performance within the existing clusters.

2.2.2 DBSCAN

We utilized the **DBSCAN** (Density-Based Spatial Clustering of Applications with Noise) algorithm to create clusters. Unlike traditional clustering methods, DBSCAN identified clusters based on data density, effectively partitioning the dataset into areas of varying data concentration while isolating noise points. We have imported the necessary library `DBSCAN` from sklearn.cluster library.

We performed a grid search on the hyperparameters `eps` (neighborhood radius) and `min_samples` for the DBSCAN algorithm. The explored values were as follows:

$$\texttt{eps} = [0.15, 0.25, 0.35, 0.5, 1.0, 2.0, 5.0, 10.0, 15.0] \quad \texttt{min\_samples} = [3, 5, 7, 10, 15]$$

After the grid search, the optimal hyperparameters were determined to be `eps` $= 15$ and `min_samples` $= 3$. Subsequently, we executed the DBSCAN algorithm using these optimal parameters to obtain the clustering results. At this stage, we examined the cluster results for both the dataframe normalized with MinMaxScaler and the one normalized with PCA. We obtained silhouette scores of 0.27 and 0.18, respectively. Consequently, we observed that the clustering performed better on the dataset normalized with MinMaxScaler. Some of the plots related to the conducted analyses are included in Figure 9 below.

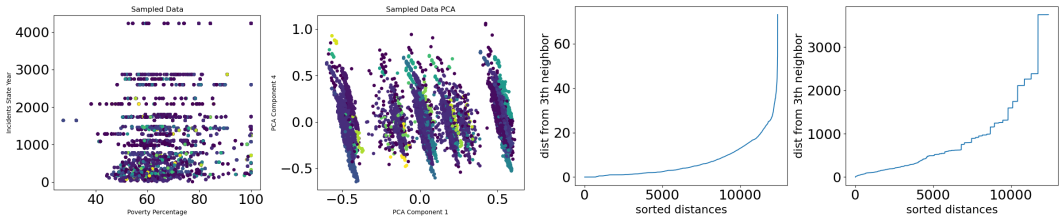


Figure 9: Results of the clustering analysis carried out with DBSCAN on MinMax and PCA.

2.2.3 Hierarchical Clustering

We incorporated **Hierarchical Clustering** into our analysis, applying the method to both the datasets normalized with MinMaxScaler and PCA. Hierarchical Clustering is a method that organizes data into a hierarchical tree-like structure, capturing relationships between data points. It iteratively merges or splits clusters based on the similarity or dissimilarity between elements, creating a hierarchy of clusters.

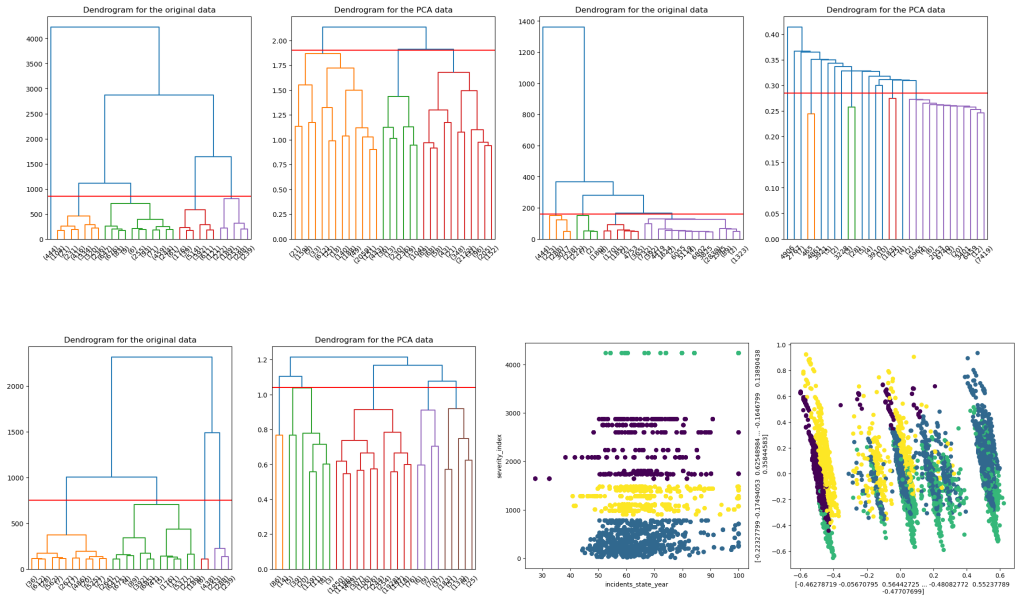


Figure 10: Results of the hierarchical clustering with complete and single linkage methods (page above) and average and ward linkage methods (up here) on MinMax and PCA.

Following a search for the appropriate number of clusters, we explored various configurations using silhouette score as the evaluation metric. We experimented with different linkage approaches, including **simple linkage, average linkage, complete linkage** and **ward linkage** methods, while maintaining the Euclidean metric. The final results are presented in Table 7 and tell us that the best method is the ward method.

Linkage Method	**Best Silhouette Score (MinMax)**	**Number of Clusters**
Complete	0.6055	7
Single	0.6066	3
Average	0.6514	5
Ward	0.665	8
Linkage Method	**Best Silhouette Score (PCA)**	**Number of Clusters**
Complete	0.2857	4
Single	0.3265	3
Average	0.3464	3
Ward	0.413	10

Table 7: Results of Hierarchical Clustering with Different Linkage Methods.

2.2.4 X-Means

We explored the implementation of **X-Means** from the Pyclustering library, using the **Bayesian Information Criterion (BIC)** as the splitting criterion. We set the minimum number of clusters to 2 and allowed them to vary up to a maximum of 20. Centroids

were initialized using the `K-Means++` heuristic. We ran the algorithm using the MinMax normalized dataframe. Despite promising results, we did not delve further into the method due to its high computational cost.

2.2.5 Clustering results

In general, we observed that the dataset normalized with MinMaxScaler outperforms the one normalized with PCA (which, on the other hand, assists in achieving similar results using fewer features) across all clustering algorithms, except for KMeans, where PCA performs better. The dataset normalized with StandardScaler yields the least favorable results.

The most promising clustering outcomes were achieved with hierarchical clustering, particularly by employing the ward linkage method. PCA also exhibits good performance, notably in KMeans. Unfortunately, we couldn't thoroughly explore X-Means, which showed potential for enhancing our results.

2.3 Other Analysis

Finally, we close this part by showing some other analyses that were performed on the dataframes and in which the application of the clustering method was necessary.

2.3.1 Latitude and Poverty Percentage

One of the initial analyses conducted focused on the attribute pair exhibiting the highest correlation, namely `latitude` and `povertyPercentage`. We performed the analyses on a subset of columns of interest, clustering based on latitude to identify 5 distinct clusters (the number of clusters suggested by the elbow method). We grouped the data according to the clusters and analyzed the results. As depicted in Figure 11, the outcomes reveal a clear trend among the central clusters, representing all U.S. states except Hawaii and Alaska: poverty tends to increase as latitude decreases (moving south). However, the lateral clusters, representing only Hawaii and Alaska individually, act as outliers and disrupt the aforementioned trend. This can be explained by their significant geographical distance from the other 4 clusters and their minimal population representation (further details in the notebook).

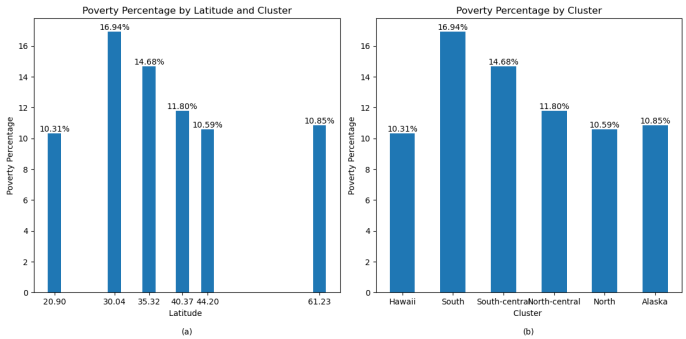


Figure 11: Results of the analysis between latitude and povertyPercentage showed by bar plots.

2.3.2 Population and Incident City Ratio

We also delved into the relationship between the `population` column and a newly defined column: `ratio_incidents_city`, i.e., the total number of incidents relative to the population of the city. We performed clustering based on population, extracting four distinct

clusters (town, small city, city, big city). The results clearly indicate an inversely proportional trend: as the population increases, the incident ratio per inhabitant tends to decrease. The results of this analysis are presented in the Figure 12 below.

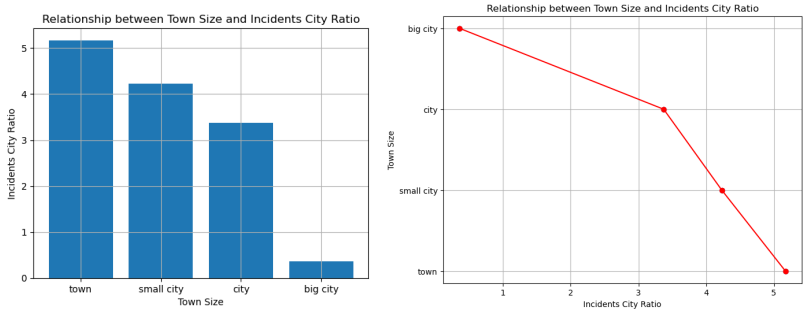
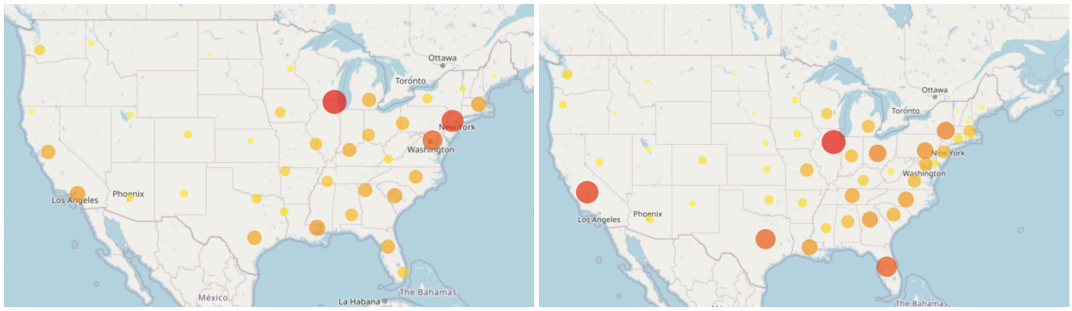


Figure 12: How the ratio of incidents city varies with population.

2.3.3 Geo Visualization

Finally, we visually presented, using the Plotly library, how the number of incidents varies across states and cities. For cities, we performed clustering and found a meaningful number of clusters, 42, with the help of the Elbow method. For states, we grouped incidents by state. From Figure 13, it can be observed that Chicago has the highest number of incidents, and the states with the most incidents are California, Florida, and Illinois.



Clustered cities. States.

Figure 13: Number of incidents for clustered cities and states.

3 Predictive Analysis: Classification

In this classification section, our goal is to predict whether at least one person has been killed in an incident. We will explore various classification algorithms to achieve this predictive task.

3.1 Dataset Preparation

In the data preparation phase, we **imported the datasets** created in the initial section of our analysis. Subsequently, we segregated the columns into categorical and numerical features. To facilitate the classification task, we **transformed the categorical columns into numerical** representations using the `discretize_data()` function.

A pivotal step involved the **creation of a new label**, `isMurder`, to discern whether at least one person is killed in a given incident. To prevent potential leakage of information into the

classification results, we proceeded to eliminate the columns `n_killed` and `severity_score`, as they contained details about individuals who had died in incidents.

3.1.1 Score Metric

At this stage, we **balanced the dataset** to ensure an equal number of occurrences for the `isMurder` label. This balancing step is crucial to prevent potential biases in the model due to imbalanced class distributions. Following this, we could use **accuracy as our primary evaluation metric** for the classification task, considering that it provides a comprehensive measure of the model's overall performance in predicting both classes, `isMurder` and non-`isMurder`.

3.1.2 Train-Test Split

At this point, we **prepared a list of labels** by populating it with the values from the `isMurder` column. Subsequently, we removed the `isMurder` column from the dataframe. For the classification task, we extracted a **training set** and a **test set** from our dataframe using the `train_test_split()` function from sklearn.model_selection library. We allocated 70% of the data to the training set and 30% to the test set, incorporating the previously created labels for `isMurder` as the target variable. Due to time constraints, we did not create and did not work with a validation set.

3.1.3 Utility Functions

Throughout our analysis, we developed several **utility functions** to facilitate the evaluation and interpretation of various classification methods. These functions serve a range of purposes, from printing classification scores and plotting confusion matrices to analyzing classification results, such as identifying misclassifications and more. These utilities play a crucial role in efficiently assessing and understanding the performance of different classification algorithms.

3.2 Methods

3.2.1 K-Nearest Neighbors

In our analysis, the initial algorithm we employed was the **k-Nearest Neighbors** (k-NN) classifier. To optimize its performance, we conducted a **grid search** on the training set, using `GridSearchCV()` function and exploring different combinations of parameters. Specifically, we varied the number of neighbors (`n_neighbors`) in the range from 1 to 15, utilized two distance metrics (`euclidean` and `manhattan`), considered three different algorithms (`ball_tree`, `kd_tree` and `brute`) and considered two weighting schemes (`uniform` and `distance`). Following this grid search, the most successful configuration was found to be using the Manhattan distance metric with 15 neighbors, using the `ball_tree` algorithm and employing distance-based weighting. This setup resulted in an **accuracy** of 0.65.

Class	Precision	Recall	F1-Score	Support
0	0.66	0.69	0.68	2631
1	0.68	0.65	0.66	2631
Accuracy			0.67	5262
Macro Avg	0.67	0.67	0.67	5262
Weighted Avg	0.67	0.67	0.67	5262

Table 8: Classification Results using k-NN.

Subsequently, we fitted the k-NN classifier with these optimized parameters and assessed its performance on the test set. The results are presented in Table 8. Additionally, we

generated visualizations to provide insights into the functioning of our classifier, offering a deeper understanding of its decision-making process.

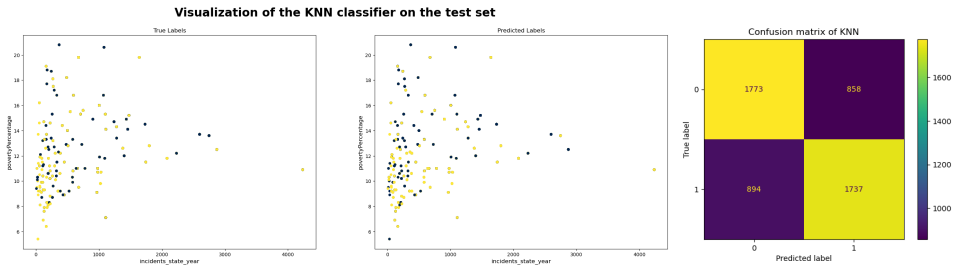


Figure 14: Scatter plot and confusion matrix of the k-NN method.

3.2.2 Radius Neighbors

The next algorithm we explored is the **Radius Neighbors Classifier**. This method, akin to k-NN, is a non-parametric algorithm used for classification. The primary difference lies in the way it selects neighbors. While k-NN considers a fixed number of neighbors, the Radius Neighbors Classifier considers all points within a specified radius.

For our analysis, we performed a grid search similar to the one conducted for k-NN, but with the substitution of the `radius` parameter in place of the `n_neighbors` parameter. The optimal hyperparameters were determined to be: metric - `euclidean`, radius - 7, weights - `uniform`, algorithm - `ball_tree`, resulting in an **accuracy** of 0.54. Following the grid search, we fitted the classifier with these parameters and evaluated the results on the test set. The obtained results were inferior to those of the k-NN algorithm.

3.2.3 Support Vector Machine

We also delved into the application of **Support Vector Machine** (SVM) for classification. Due to the computational intensity, we refrained from an exhaustive grid search. Instead, we utilized the `SVC()` function from the `sklearn.svm` module, focusing on two different kernels: `radial_basis_function` (rbf) and `sigmoid`.

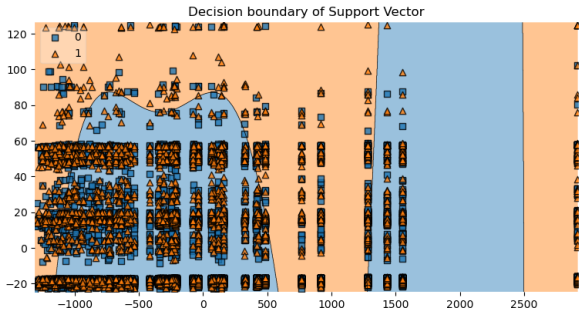


Figure 15: Decision boundary of SVM method.

The chosen hyperparameters for both kernels were `gamma = scale` and `C = 0.5`. After fitting the models, we evaluated their performance on the test set. The results indicated that the SVM with the rbf kernel outperformed the sigmoid kernel but lagged behind the k-NN algorithm.

3.2.4 Feed-Forward Neural Network

Moving on to explore classification with neural networks, we initially employed a simple **Feedforward Neural Network**. Following our standard practice, we conducted a grid search with the following parameters: `batch_size` = [500], `epochs` = [100], `optimizer` = ['SGD', 'adam', 'adamax'], `activation` = ['softmax', 'relu', 'sigmoid', 'linear'], `dropout_rate` = [0.2], `neurons` = randint(5, 20), `loss` = ['binary_crossentropy', 'mean_squared_error', 'categorical_crossentropy'].

The optimal parameters we discovered were `activation`: 'linear', `dropout_rate`: 0.2, `batch_size`: 500, `epochs`: 100, `loss`: 'mean_squared_error', `neurons`: 7, `optimizer`: 'adam', resulting in an accuracy of 0.6. After fitting the model, we evaluated its performance on the test set and observed results similar to those obtained with the SVM.

3.2.5 Multi-Layer Perceptron

Moving on to larger neural networks, we explored the **Multi-Layer Perceptron** (MLP). In this case, we opted to use the dataframe normalized with `MinMaxScaler`, as it outperformed the non-normalized dataframe. The parameters for our grid search were defined as follows: `hidden_layer_sizes`: [100, 200], `activation`: ['tanh', 'relu', 'identity', 'logistic'], `solver`: ['sgd', 'adam', 'lbfgs'], `alpha`: [0.0001, 0.01], `learning_rate`: ['constant','adaptive', 'invscaling'].

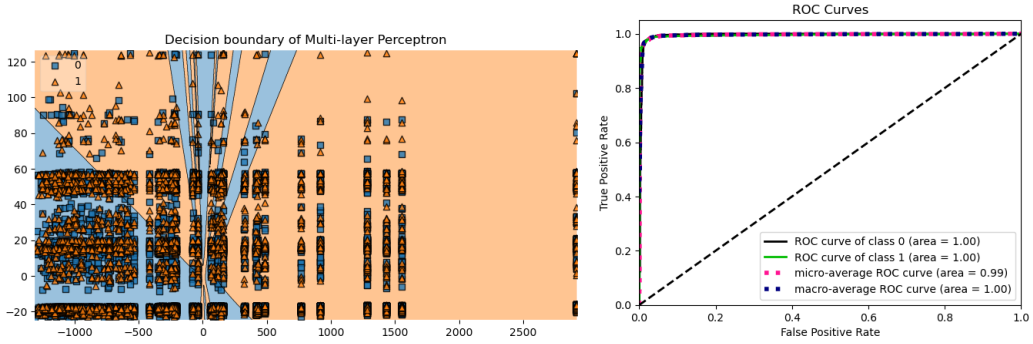


Figure 16: Decision boundary and ROC curves of MLP method.

The best setting parameters we found were: `activation`: 'tanh', `learning_rate`: 'constant', `alpha`: 0.0001, `hidden_layer_sizes`: 100, `solver`: 'sgd'. After fitting the model with these parameters, we evaluated its performance on the test set, achieving significantly higher accuracy compared to previous methods, as we can see in Table 9.

Class	Precision	Recall	F1-Score	Support
0	0.96	0.99	0.98	2631
1	0.99	0.96	0.98	2631
Accuracy			0.98	5262
Macro Avg	0.98	0.98	0.98	5262
Weighted Avg	0.98	0.98	0.98	5262

Table 9: Classification Results on test set using Deep Neural Networks.

3.2.6 Gaussian Naive Bayes

Transitioning to **Naive Bayes methods**, we first delved into the **Gaussian Naive Bayes classifier**. This algorithm, based on Bayes' theorem, assumes independence among features and performs well with continuous data. After fitting the model on the training set, achieving

an accuracy of 0.71, we evaluated its performance on the test set, where it demonstrated effective classification, as we can see in Table 10.

Class	Precision	Recall	F1-Score	Support
0	0.68	0.78	0.73	2631
1	0.74	0.64	0.69	2631
Accuracy			0.71	5262
Macro Avg	0.71	0.71	0.71	5262
Weighted Avg	0.71	0.71	0.71	5262

Table 10: Classification Results using Gaussian Naive Bayes.

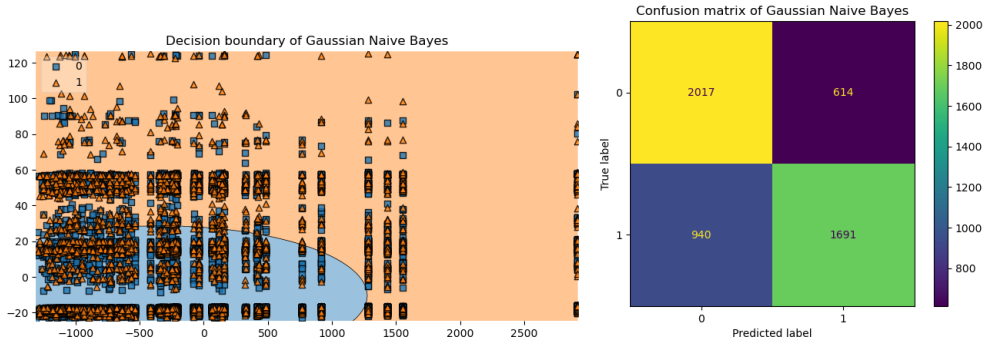


Figure 17: Decision boundary and confusion matrix of Gaussian Naive Bayes method.

3.2.7 Multinomial Naive Bayes

Subsequently, we explored the **Multinomial Naive Bayes** classifier, which is suitable for discrete data and often used for text classification. The key difference from the Gaussian Naive Bayes lies in the type of distribution assumed for the features. We conducted a grid search on the parameter α with values [0, 1, 2, 3, 5, 6, 7, 8, 9]. The best setting was found with α=0. However, upon fitting the model and evaluating its performance on the test set, we observed results significantly worse than those obtained with Gaussian Naive Bayes and slightly better than those achieved with SVM.

3.2.8 Decision Tree

Moving on to the Decision Tree classifier, a supervised machine learning algorithm that makes decisions based on recursive partitioning of the input data. In this method, we conducted a grid search on the training set with the following parameters: `dt_max_depth`: [2, 3, 5, 6, 7, 10, 12, None], `dt_min_samples_split`: `randint(2, 51)`, `min_samples_leaf`: `randint(1, 51)`, `max_features`: [None, 2, 3, 4, 5], `splitter`: ["best", "random"], and the last `criterion`: ["entropy", "gini"].

Class	Precision	Recall	F1-Score	Support
0	0.94	0.97	0.95	2631
1	0.97	0.94	0.95	2631
Accuracy			0.95	5262
Macro Avg	0.95	0.95	0.95	5262
Weighted Avg	0.95	0.95	0.95	5262

Table 11: Classification Results using Decision Tree.

The best parameters found were `criterion`='gini', `max_depth`=7, `max_features`=None, `min_samples_leaf`=3, `min_samples_split`=28, `splitter`='best', resulting in an accuracy of 0.95. Upon fitting the model with these parameters on the training set and evaluating its performance on the test set, we achieved results inferior only to the multi-layer perceptron.

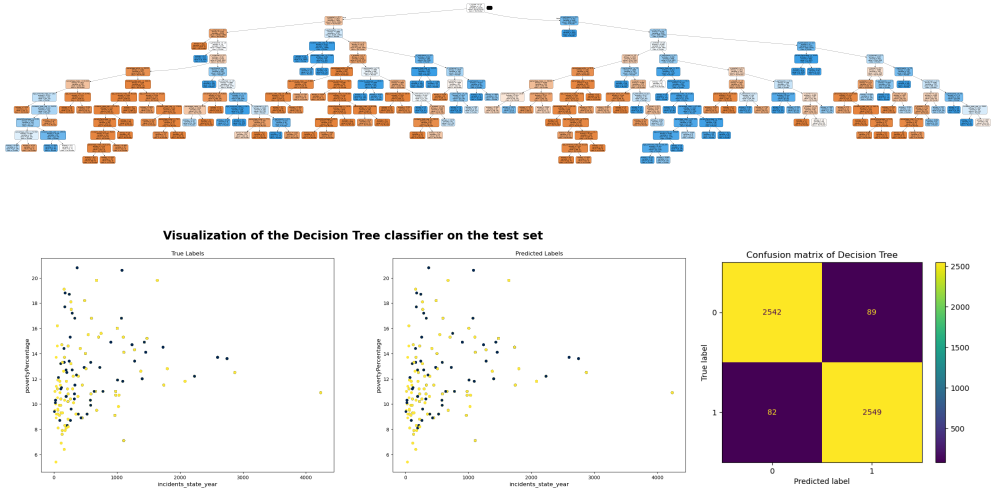


Figure 18: Decision tree, scatter plot and confusion matrix obtained using the Decision Tree method.

3.2.9 Random Forest

Transitioning to ensemble methods, we employed **Random Forests**, a powerful ensemble learning technique. This method involves constructing a multitude of decision trees and merging them to enhance predictive accuracy and control overfitting. For the grid search, we considered the following parameters: `max_depth`: [2, 3, 5, 6, 7, 10, 12, None], `max_features`: [None, "sqrt", "log2"] + `list(range(1,len(train_set.iloc[0])`, `bootstrap`: [True, False], `min_samples_split`: `randint(10, 51)`, `min_samples_leaf`: `randint(10, 51)`, `criterion`: ["entropy", "gini"], `class_weight`: ['balanced', 'balanced_subsample', None].

The best parameters found were `bootstrap`: False, `class_weight`: 'balanced_subsample', `max_depth`: None, `max_features`: 6, `min_samples_split`: 35, `min_samples_leaf`: 27, `criterion`: 'entropy'. Upon fitting the model with these parameters and evaluating it on the test set, we obtained results slightly inferior to those of the Decision Tree.

Class	Precision	Recall	F1-Score	Support
0	0.95	0.93	0.94	2631
1	0.93	0.95	0.94	2631
Accuracy			0.94	5262
Macro Avg	0.94	0.94	0.94	5262
Weighted Avg	0.94	0.94	0.94	5262

Table 12: Classification Results using Random Forest.

3.2.10 Ruled Based

A **Rule-Based Classifier** is a type of model that makes predictions based on a set of rules derived from the input data. One such algorithm we explored is `RIPPER` (Repeated Incremental Pruning to Produce Error Reduction), available in the Wittgenstein library. RIPPER is known for its ability to generate comprehensible rule sets while maintaining

competitive classification performance. To tune the model, we performed a grid search on the parameters `prune_size`: [0.5, 0.6, 0.7, 0.8, 0.9] and `k`: [1, 3, 5, 7]. The optimal combination was found to be `k`: 1 and `prune_size`: 0.5.

After fitting the model on the training set, we evaluated its performance on the test set. Unfortunately, the results were significantly lower compared to other explored methods, making it the least effective classifier among those we experimented with. While rule-based classifiers can provide interpretable rules, RIPPER's performance on this specific dataset may not be as competitive as more complex models like neural networks or ensemble methods.

3.2.11 Voting Classifier

We then proceeded to work with the **Voting Classifier**, an ensemble method that combines predictions from multiple individual classifiers. In our case, we utilized the following estimators within the `VotingClassifier`: SVM, Decision Tree, Gaussian Naive Bayes, Random Forest, and MLP. After fitting the model, we evaluated its performance on the test set, achieving outstanding results showed in Table 13.

Class	Precision	Recall	F1-Score	Support
0	0.97	0.96	0.97	2631
1	0.96	0.97	0.97	2631
Accuracy			0.97	5262
Macro Avg	0.97	0.97	0.97	5262
Weighted Avg	0.97	0.97	0.97	5262

Table 13: Classification Results using the Voting Classifier.

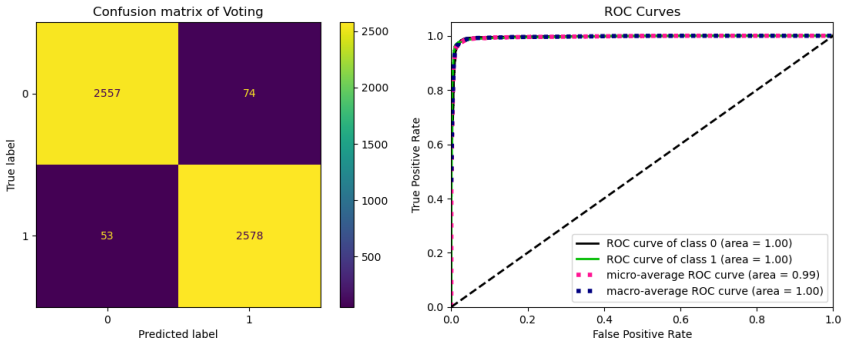


Figure 19: Confusion matrix and ROC curves of Voting Classifier method.

3.2.12 Conclusion

In conclusion, we explored various classification models. The results of final accuracies on the test data show that the Multi-Layer Perceptron model achieved the highest performance with a value of 0.977. Following closely are the Voting Classifier and the Decision Tree with accuracies of 0.976 and 0.975, respectively. Random Forest reached a good level of accuracy at 0.949. Instead, the simple Neural Network (NN) and the Rule-Based Classifier provided the lowest performance, with an accuracy around 50%. All the results are presented in Table 14, sorted starting from best method.

Despite variations in performance, it is interesting to note that no classifier achieved extremely low results, indicating that the overall classification of the dataset was generally

effective across all explored models. Therefore, we can be satisfied with the results of our classification.

Classifier	Accuracy
Multi-layer Perceptron	0.977
Voting Classifier	0.976
Decision Tree	0.975
Random Forest	0.949
Gaussian Naive Bayes	0.705
K-Nearest Neighbors	0.667
Radius Neighbors	0.633
Support Vector Machine	0.533
Multinomial Naive Bayes	0.525
Neural Network (NN)	0.506
Rule-Based Classifier	0.500

Table 14: Final Accuracies of Classification Models on Test Data.

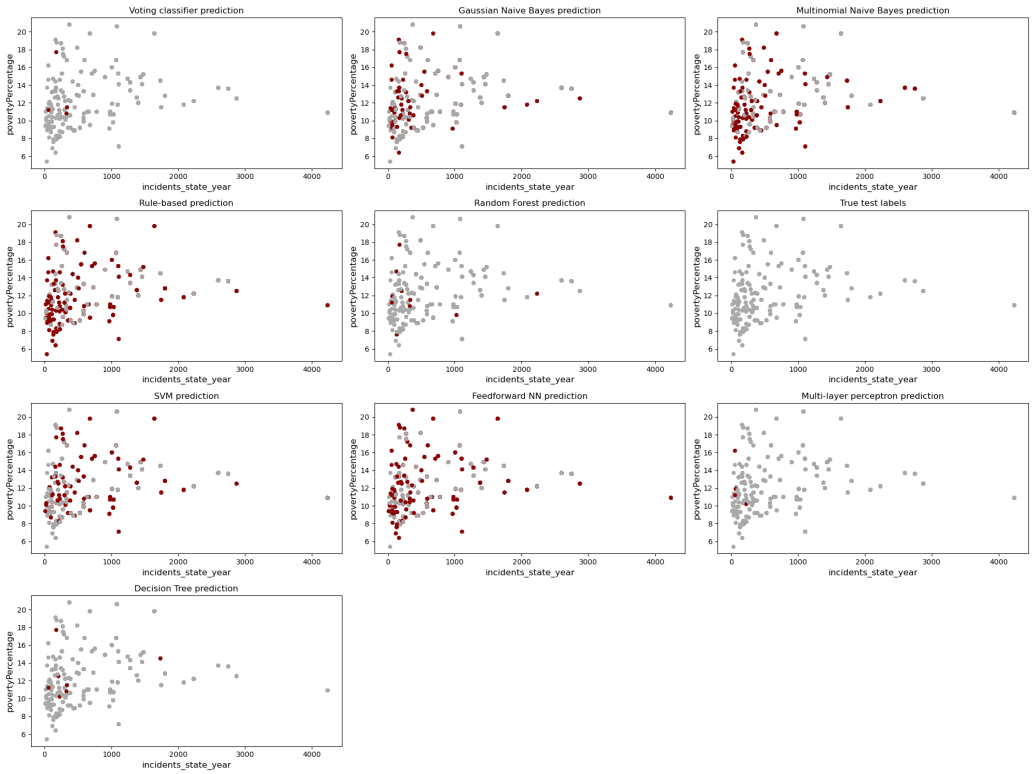


Figure 20: Scatter plot of how the various classifiers works in two dimensions.

4 Time Series Analysis

Time Series Analysis involves studying data points collected or recorded over time. In this section, we explore the temporal dimension of the incidents dataset, focusing on events occurring between 2014 and 2017. By creating time series for each city, we aim to uncover patterns, trends, and variations in incident severity scores over the specified four-year period. The goal of this task is grouping similar cities through the use of the created time series, based on the score that we define above.

4.1 Data Preprocessing

We imported the datasets created in the initial phase. We focused on the years 2014, 2015, 2016, and 2017, selecting only the relevant columns. To ensure meaningful analyses, we filtered out cities with fewer than 15% of the total weeks containing incidents over the four-year period. The new dataframes created are called `filtered_incidents`.

4.1.1 Incident Severity Score

To quantify the severity of incidents, we compute a weekly `incident_severity_score` using the following equation:

$$\texttt{severity\_score} = \frac{\texttt{n\_participants}}{2}\left(3 \cdot \texttt{n\_injured} + 2 \cdot \texttt{n\_arrested} + -2 \cdot \texttt{n\_unharmed}\right)$$

This score aims to capture the incidents' impact relative to the population, providing insights into the severity of incidents on a weekly basis for each city throughout the specified years. We used this index as an attribute during the first part of the section for clustering. In the second part, related to shapelets, we employed it for predictions, aiming to group cities based on the incident rate.

We have thus incorporated this index into the `filtered_incidents` dataframe. Additionally, we added the `isKilled` column to the `filtered_incidents` dataframe, maintaining a balanced distribution between true and false values. Subsequently, we created two additional dataframes, in which each row is a time series, to work with during this section.

- The first one, named `filtered_cities`, has the cities from the dataframe as row indices, the tuple (`year`, `week`) as column indices, and the corresponding severity scores as data.
- The second dataframe, named `time_series_analysed`, is identical to `filtered_cities`, but in this case, we refrain from scaling as done in the other dataframes (refer to Section 4.1.4). For this dataframe, we manually apply **Offset Translation**, **Amplitude Scaling**, and also compute the **Linear Trend**.

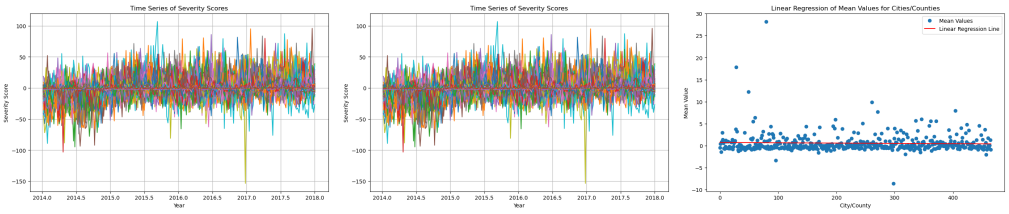


Figure 21: Time series after applying offset translation and amplitude scaling, and linear trend.

4.1.2 Trends Removal

In our analysis, we employ the **Augmented Dickey-Fuller test** to assess the stationarity or non-stationarity of our time series. By setting a threshold of 0.05 on the p-value returned by the test, we identify all time series that exhibit non-stationarity. This outcome aligns with our expectations, confirming that our time series lack trends and adhere to linearity, appearing parallel to the x-axis.

4.1.3 Noise Removal

In the context of our analysis, we aim to perform **noise removal**, a process designed to eliminate unwanted variations or irregularities in our time series data. To identify the

optimal sliding window for noise removal, we employed the **mean of the sums of absolute differences (SAD)** between the original time series and its smoothed counterpart. Utilizing the elbow method, we determined the most suitable window size - one that strikes a balance between effective noise reduction and preserving the original time series characteristics. The selected window size was found to be 5. Subsequently, we applied the `denoiser()` command to effectively eliminate noise from our time series data.

4.1.4 Scaling

To conclude the preprocessing phase, we performed scaling on the time series data. For this task, we opted for the `Mean-Variance` scaler to standardize the time series, ensuring consistency in their statistical properties.

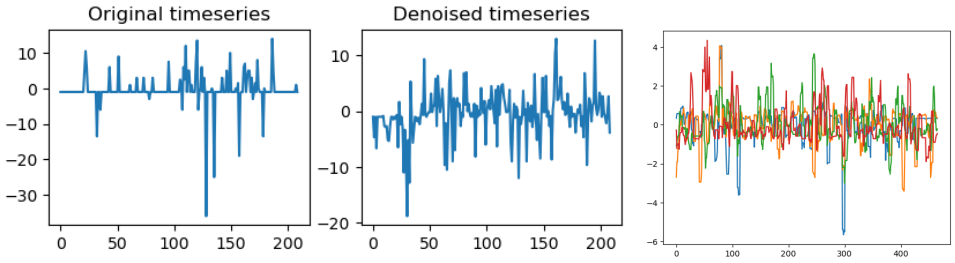


Figure 22: Original time series, denoised time series and scaled time series.

4.2 Clustering Timeseries

Before moving on to the clustering of time series, we defined three utility functions to assist us in the plots of the next section. We will then explore three different clustering algorithms, all on the time series present in the `filtered_cities` dataframe.

4.2.1 Shape-Based Clustering

We employed **Shape-Based clustering**, a method that grouped time series based on their shapes. The chosen metric was the **Euclidean distance**. To determine the optimal number of clusters k, we applied the elbow method on the sum of squared errors (SSE) within the range from 2 to 21.

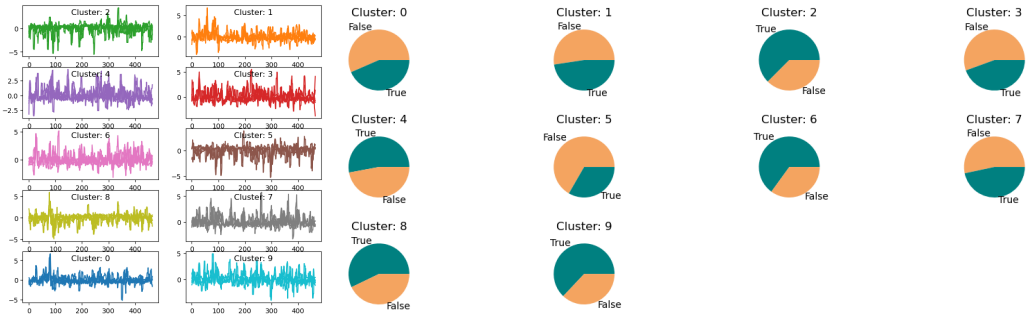


Figure 23: Clusters obtaneid using Shape-Based method with Euclidean distance.

The best k was found to be 10, we ran the algorithm with this value e we obtained a Silhouette Score of 0.1. Subsequently, we presented the results of the algorithm, presented in Figure 23.

Dynamic Time Warping

We then explored the same as before, but with `Dynamic Time Warping` (DTW) as metrics. DTW is a technique used to measure the similarity between two sequences with different lengths. Due to its computational cost, we were unable to perform a grid search. Nevertheless, we manually explored various parameters, and the ones we considered optimal was `n_clusters`=10. Subsequently, we executed the algorithm with these parameters and presented the obtained results, that are showed in Figure 24.

Figure 24: Clusters obtaneid using Dynamic Time Warping.

4.2.2 Feature Based Clustering

We also worked with **Feature-Based clustering**, which involves defining a set of features to extract from the time series. Subsequently, we applied standard clustering techniques using tabular data. From each time series, we extracted features such as *Average, Standard Deviation, Variance, Median, 10th Percentile, 25th Percentile, 50th Percentile, 75th Percentile, 90th Percentile, Interquartile Range, Coefficient of Variation, Skewness* and *Kurtosis*.

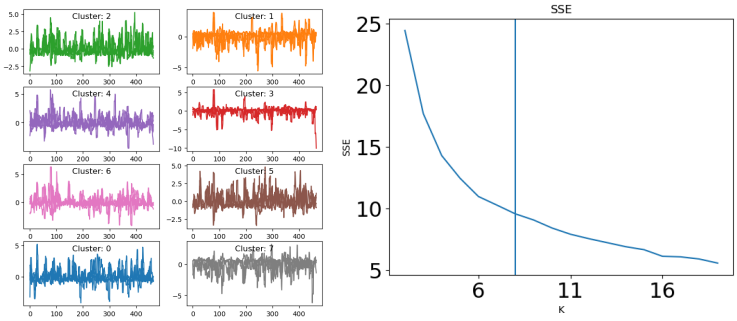


Figure 25: Clusters obtaneid using Feature Based method and elbow method computed.

Following feature extraction, we normalized the data using `MinMax` scaler. The normalized data were then utilized to perform k-means clustering. Initially, we determined the appropriate k value using the elbow method, and we found that k=8 yielded the best results. Subsequently, we executed the algorithm with this optimal k and presented the obtained results.

Hierachical

For the **hierarchical** we tried all the variation we already tried in the Section 2.2.3.We use both `Standard` and `MinMax` normalization for this part. At the conclusion of our exploration of various clustering methods, performed on the two dataframes employing different normalizations, we calculated and examined the silhouette scores for each method. Remarkably,

the **ward linkage** method with `MinMax` normalization consistently exhibited the highest silhouette scores. Motivated by these findings, we opted to utilize this combination to create hierarchical clusters.

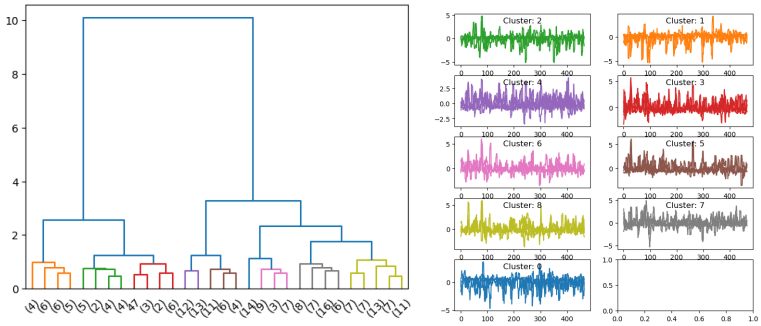


Figure 26: Clusters obtaneid using hierarchical method, with ward linkage and MinMax scaler.

4.2.3 Compression Based clustering

We also delved into **Compression-Based clustering**, a technique involving the use of the **Piecewise Aggregate Approximation** (PAA). The PAA approximates the original time series by representing it with piecewise lines. We used 16 as the number of segments which compress the timeseries, because in preliminary trials we found that it provided a good and representative compression.

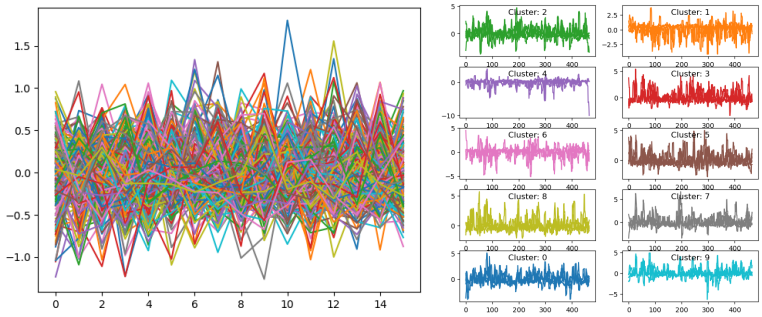


Figure 27: Clusters obtaneid using Compression Based method.

To determine the optimal number of clusters k, we employed the k-means algorithm. It was revealed that the suitable number of clusters for this method is 10, and consequently, we executed the clustering algorithm with k=10.

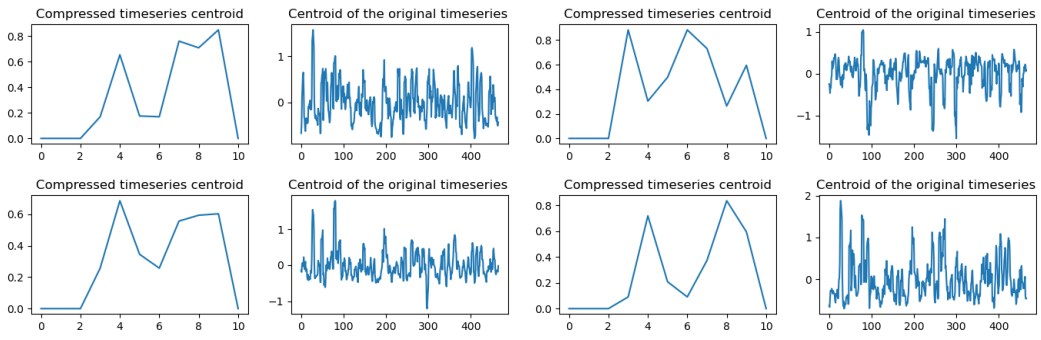


Figure 28: Some centroids obtaneid during Compression Based method.

4.3 Motifs and Anomalies

In this section we work on the `time_series_analysed` dataframe, the dataframe cleaned by hand from us in the Section 4.1.

4.3.1 Motifs Discovery

We so proceeded with **Motifs Discovery**, a technique aimed at identifying recurring patterns in time series data. For this analysis, we selected a window size of w=12. Utilizing the `matrixProfile.stomp()` command on the time series and the chosen windows, we sought to create a concise representation of similarities among subsequences within a time series. Subsequently, we detected motifs using the `motifs()` command, where we set `max_motifs`=5, and presented the identified motifs.

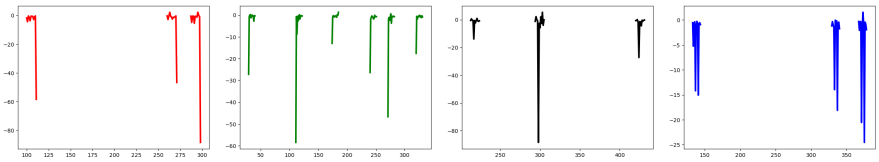


Figure 29: Motif discovered.

4.3.2 Anomalies Discovery

We then proceeded with **Anomalies Discovery**, a process aimed at identifying patterns in time series data that deviate significantly from the norm. Once again, we employed the previously calculated matrix profile numpy array, with the same window size $w = 12$. We opted for k=5, where k represents the number of discords to be discovered. Additionally, we selected `ex_zone`=3, denoting the number of samples to exclude, set to Inf on either side of a found discord. The anomalies were computed using the `discord()` command, and the results were subsequently printed.

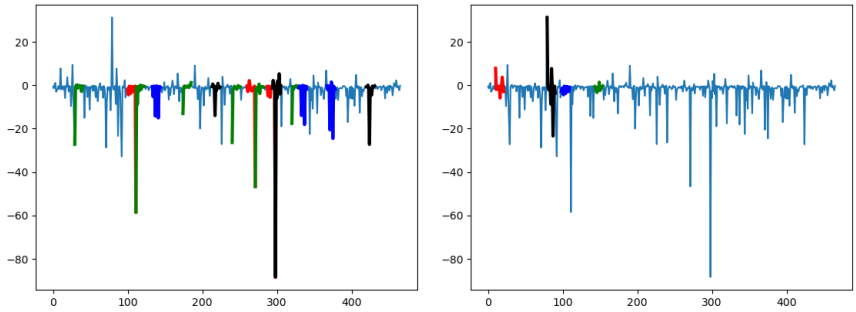


Figure 30: Motif discovered and anomalies discovered.

4.4 Shapelet

In the final section of the report, we delve into **Shaplets**, which are discriminative subsequences within time series data. These shaplets are computed on the `filtered_cities` dataframe and the labels that we saved in the preprocessing part.

The initial step involves categorizing all time series into two classes: *not killed* and *killed*. The **Grabocka method** is then employed to identify the optimal number of shapelets for each shaplet size. After thorough exploration, we determine that the optimal shaplet size is

12, with 4 shaplets identified for this size.

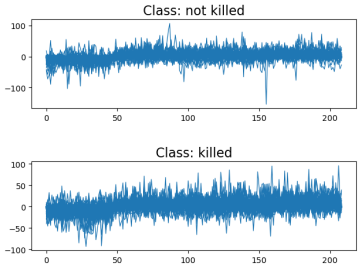


Figure 31: Classes used for shapelet.

With this information at hand, a Shapelet model is trained, comprising a time series transformation and a logistic regression layer. The optimization of Shapelet coefficients and logistic regression weights is achieved through the Adam optimizer, with a maximum of 15 training epochs and a regularization factor of 0.001. To evaluate the model's performance, we employ a stratified train-test split of 75-25. The resulting accuracy metrics are presented in the Table 15.

Class	**Precision**	**Recall**	**F1-Score**	**Support**
False	0.57	0.47	0.51	58
True	0.56	0.66	0.60	59
Accuracy			0.56	117
Macro Avg	0.57	0.56	0.56	117
Weighted Avg	0.57	0.56	0.56	117

Table 15: Classification Results using the Shapelet Model.

In conclusion, this model allows us to analyze the time series and observe the number and position of shapelets found within each one.

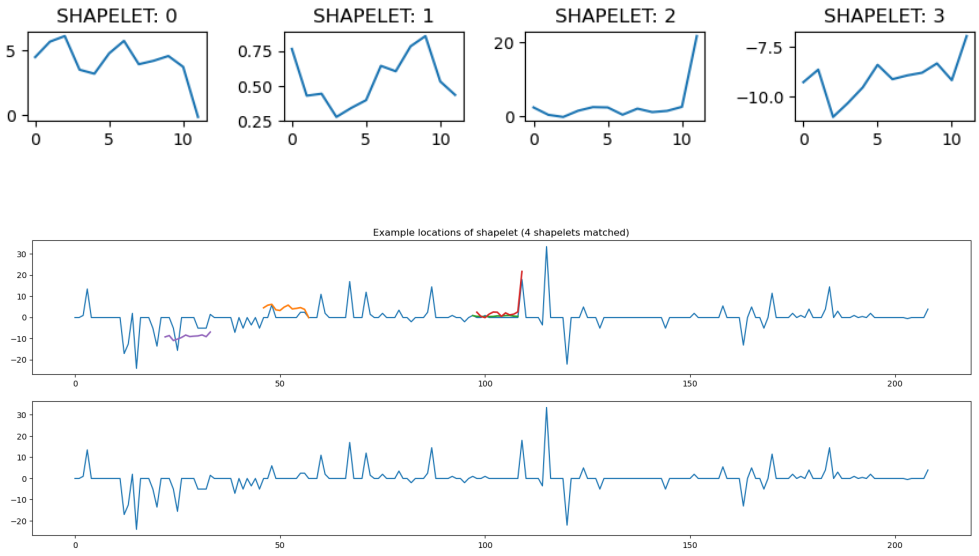


Figure 32: Shapelet and position of shapelets found.