Università di Pisa

Data Mining II - 2017

Sequential Pattern Mining, Time Series Analysis,
Classification & Outlier Detection using tools powered by Python libraries

By: Mario Shtjefni
Giuseppe Onesto

Sequential pattern mining

In order to apply sequential pattern mining techniques on the two datasets that encode 100 of Bach's chorales as sequences of numbers, the SPMF open-source library was used. Among the many SPM algorithms available at this library, SPAM was the one we chose to apply on both datasets. It enables the definition of temporal constraints and the specification of the minimum or maximum pattern length i.e. the number of events in each sequence while at the same time being one of the fastest SPM algorithms.

Top five most frequent sequences with at least five notes:

To extract the top five most frequent sequences, *minsup* i.e. the frequency threshold had to be set to ≈ 66% to filter out the less popular ones. In addition, the "Min pattern length" attribute was set to "5" to make sure that the obtained subsequences have at least five events. Using these settings, the algorithm found six subsequences but below we show only five:

```
69 -1 69 -1 69 -1 69 -1 69 -1 #SUP: 75

69 -1 69 -1 69 -1 69 -1 69 -1 69 -1 #SUP: 72

69 -1 74 -1 69 -1 69 -1 69 -1 #SUP: 68

69 -1 69 -1 69 -1 69 -1 67 -1 #SUP: 66

69 -1 69 -1 69 -1 69 -1 69 -1 69 -1 69 -1 #SUP: 66
```

The "-1" is used to separate the elements or to signify the end of the sequence.

There is a tie for fourth place shared between three generated subsequences, each of which has a support of 66 (one of them was left out to comply with the request of the exercise). As can be seen from the results, the note "69" is quite popular among the patterns. In fact, the first, second and fifth patterns only contain this note. In addition, the first pattern is a contiguous subsequence of the second one, which itself is a contiguous subsequence of the fifth one; of course, this also means that their support is greater than the latter.

For the second dataset containing the note-duration representation, the encoding is done by summing the corresponding number of each note with the product of the note's duration with 100. Even though the notes are the same, their durations may vary since they are subject to musical technicalities. Therefore, we end up with a different set of sequences. The *minsup* was brought down to ≈ 51% to get the top five most frequent ones:

```
469 -1 469 -1 469 -1 469 -1 469 -1 #SUP: 59

471 -1 471 -1 469 -1 471 -1 469 -1 #SUP: 53

471 -1 471 -1 471 -1 471 -1 469 -1 #SUP: 52

469 -1 471 -1 471 -1 469 -1 469 -1 #SUP: 51

471 -1 469 -1 471 -1 469 -1 469 -1 #SUP: 51
```

It is not easy to compare the results with the sequences generated in the previous case, due to of the different nature of the dataset. However, we can think of the first pattern for example as the same pattern obtained in the 1st place, composed of the note '69' repeated for five times with a duration of four units.

Top five most frequent contiguous sequences with at least four notes:

As in the previous case, we have to use the parameter "Min pattern length" to ensure that the generated subsequences have at least four notes. What is different this time is that they also need to be contiguous. This is done by setting the "Max gap" parameter to "1". It restricts the time difference between two consecutive elements in the sequence to be at just one unit. In other words, notes have to occur immediately after each other.

For the first dataset, the appropriate *minsup* turned out be ≈ 19%. There is a tie for fifth place, with four patterns having the same support of 19:

```
74 -1 72 -1 71 -1 69 -1 #SUP: 27

71 -1 72 -1 71 -1 69 -1 #SUP: 23

69 -1 71 -1 73 -1 74 -1 #SUP: 21

72 -1 74 -1 72 -1 71 -1 #SUP: 20

67 -1 69 -1 71 -1 72 -1 #SUP: 19
```

For the second dataset, the minimal frequency threshold to get five patterns was relatively low (≈ 0.09%):

```
474 -1 472 -1 471 -1 469 -1 #SUP: 11

471 -1 472 -1 471 -1 469 -1 #SUP: 10

467 -1 469 -1 471 -1 472 -1 #SUP: 9

473 -1 474 -1 473 -1 471 -1 #SUP: 9

474 -1 473 -1 471 -1 469 -1 #SUP: 9
```

Conclusions

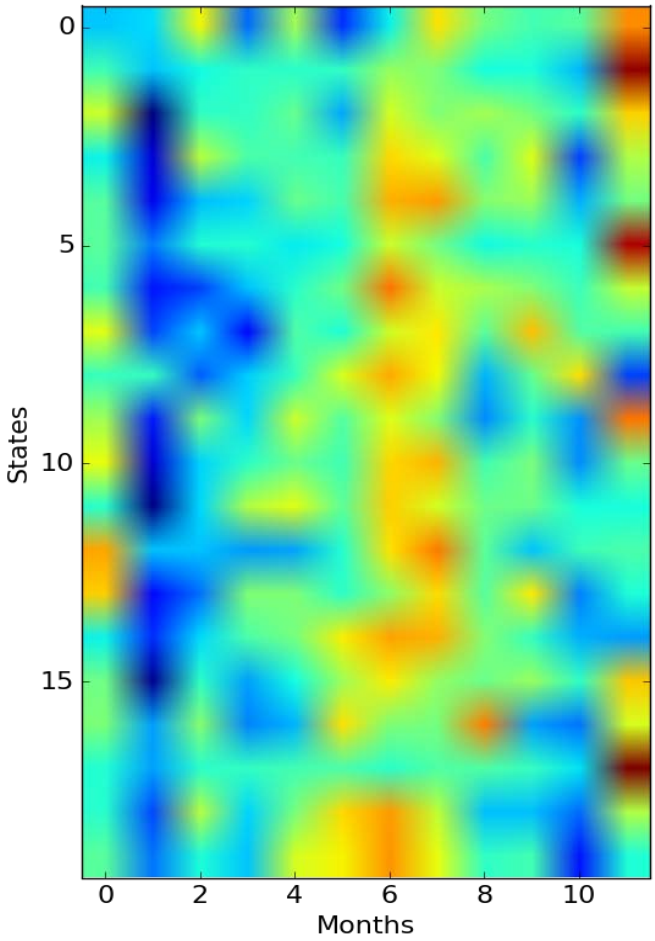
What is notable in this dataset is that the *minsup* threshold necessary to spawn the top five contiguous sequences, be it with or without duration, is significantly lower than the threshold required to generate the non-contiguous subsequences. This shows us that there are far fewer cases in which Bach utilizes the same four or more notes consecutively.

Time Series

For the analysis of the temporal features on the U.S. homicide database, we have assumed that a row represents a single murder or manslaughter case. The column named “Victim Counts” has not been taken into consideration due to its ambiguity for the information it conveys (it could be the *total* or the *additional* number of victims involved in the homicide). Either way, around 92% of values in this column are “0” i.e. *missing data*/*no-additional victims*, hence its exclusion from the study is not likely to have a significant impact on the conclusions.

We begin our analysis on by making a few general observations. Grouping the raw dataset by the attribute “Month”, we obtain the total number of homicides across 35 years for each month. A matrix representation of this aggregation for the first 15 states is shown below:

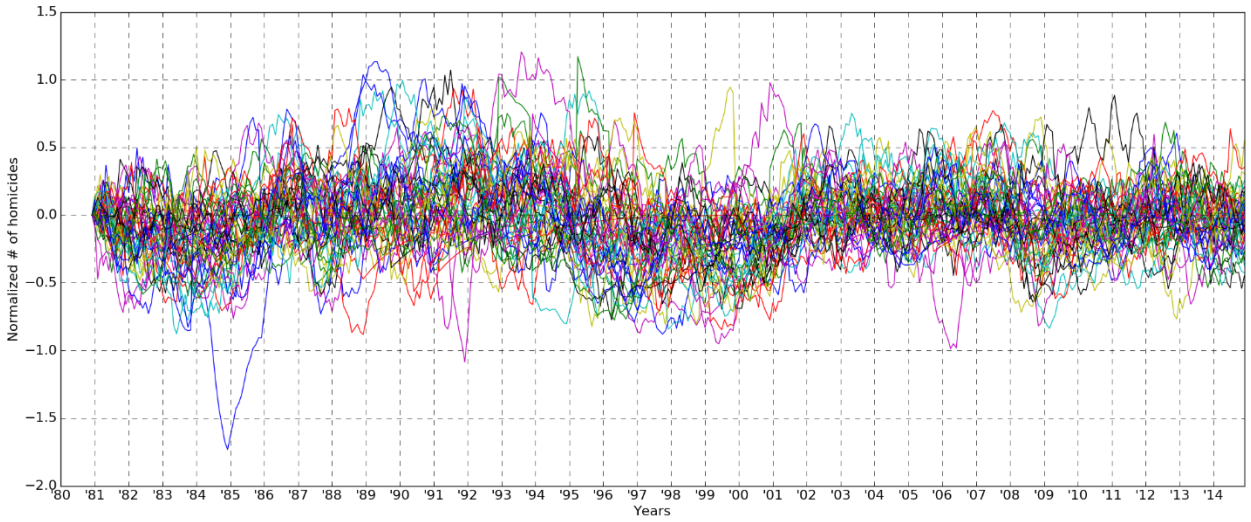
Figure 1 - Aggregation by month



Notice that in general, a higher number of murders occurred during July, August and December, as indicated by the yellow-to-orange spots while the homicide rate rather drops in the month of

February. This pattern is persistent over the majority of states. However, indications of any kind of periodicity become less evident if we group by year:

Figure 2 - Aggregation by year

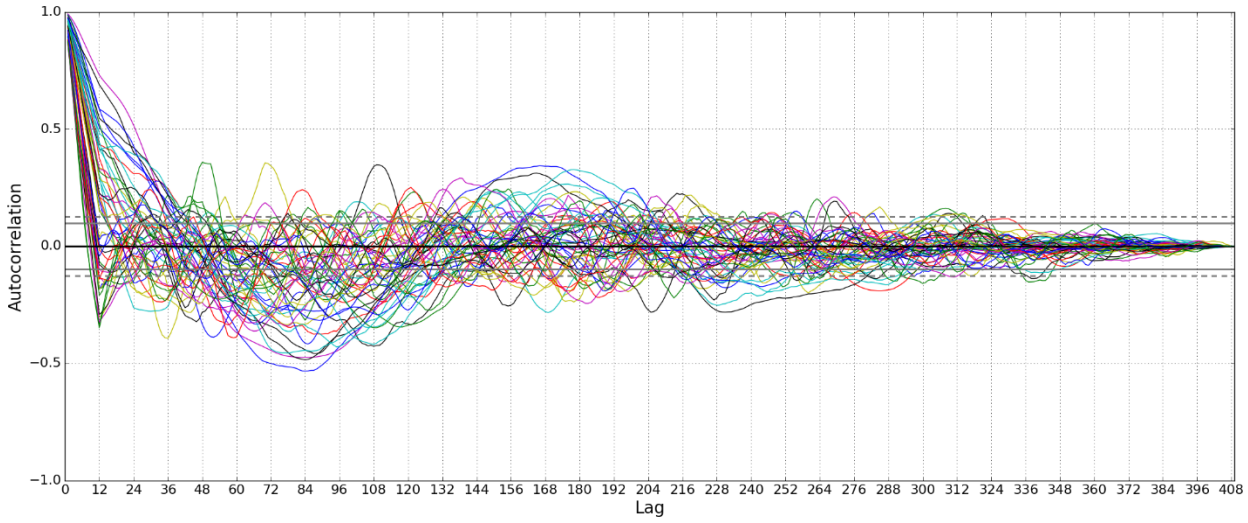


Here the data was normalized in order to create a better idea of the differences between the states and the trends present throughout the years, given than some subjects such as California, Texas or New York overshadow others in comparison, since they have the overall highest number of murders committed. We did not bother with the labels, for obvious reasons. Broadly speaking, there is a slight increase in the recorded number of victims during the early 90s, followed by a decrease of fatal cases starting from the 2000s.

Next, we grouped by year and month, obtaining 420 rows (35y ***** 12m). The new granularity enables a better understanding of the periodicity of the murders, if present, since the months are repeated year after year. From here on, we will only work with this type of grouping. The data normalization was done following the four general transformation rules: offset translation, amplitude scaling, noise and linear trend removal. In order to minimize the noise effects, we took the smoothing approach by computing the rolling mean using the last 12 values i.e. the average over the time-period of the past 1 year. In addition, we performed Augmented Dickey-Fuller tests on each of the states and found that in quite a few instances the null hypothesis that hints a presence of a unit root could not be rejected. In other words, a linear trend was present and to remove it we used the exponentially weighted moving average technique where weights are assigned to previous values with a decay factor (which we conveniently set to 12). The obtained fitting line was then subtracted from each state graph.

The autocorrelation plot of the time series is presented below. It is a tool for finding repeating patterns within a signal by comparing it with a delayed copy of itself:

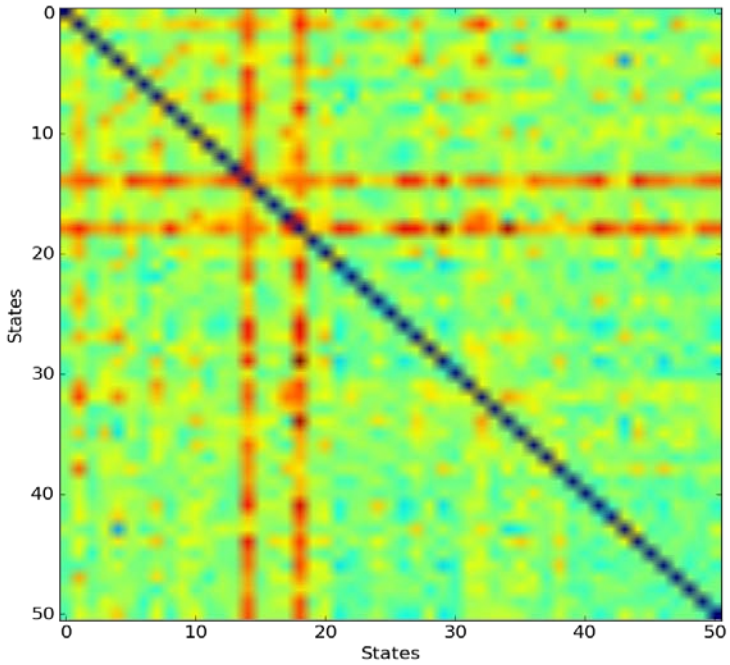
Figure 3 - Autocorrelation plot with Year-Month group by



The x-axis represents the lag in months (409 in total since the first 11 values are not regarded). Upon first view, it seems as though there is a weak seasonality that stretches beyond the yearly period. For some states, the coefficient becomes significantly non-zero during the initial 3 to 9 year lags, indicating a negative correlation with the previous period. It rises again above zero for the 11 to 15 year lags (corresponding to the early 90s). However, there is not enough evidence to suggest a strong overall periodicity in the series since for the majority of the states the autocorrelation coefficient lies between or close to the 95% confidence bands.

DBSCAN with Dynamic Time Warping

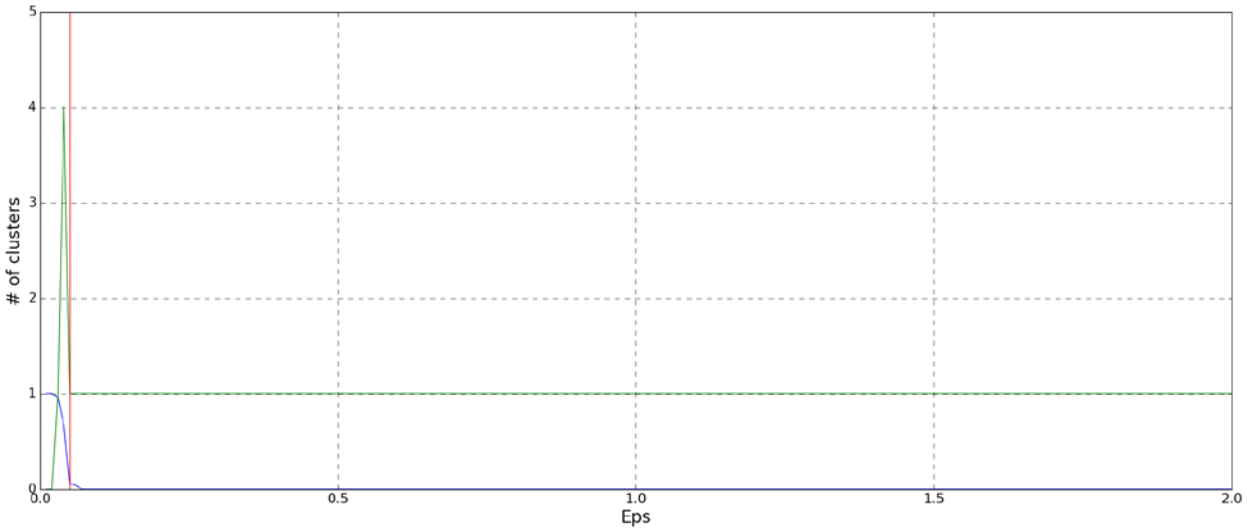
Figure 4 - Matrix representation of DTW distances



In the next step of our analysis, we aim to cluster together the states with the most similar characteristics by applying the Density Based Scan algorithm on the precomputed dynamic Time Warping distances. The above figure depicts a symmetrical matrix representation of the DTW distance between the states. The computation of these distances took about 2 hours on an Intel Core i7 processor with 8GB of available RAM. The most noticeable feature is the somewhat surprising contrast between Illinois and Louisiana and the rest of the states as indicated by the thick red lines respectively.

To find the best parameters for the DBSCAN algorithm, 200 possible *eps* i.e. radius values were plotted against the corresponding number of clusters (green line) and of noise points (blue line):

Figure 5 - DBSCAN parameter search



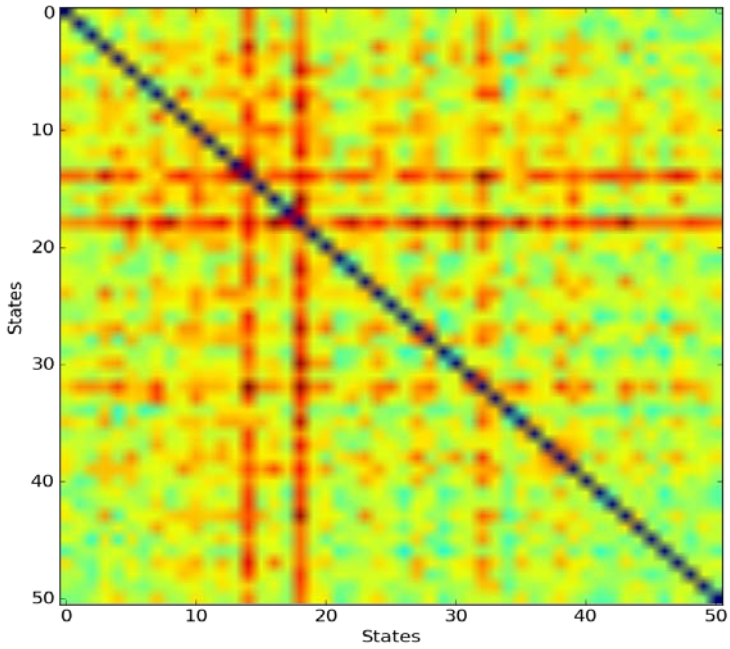
For every *eps* value smaller than 0.04 or greater than 0.06 DBSCAN would either classify all points as outliers or put them into a single cluster. A radius of exactly 0.06 provided the highest silhouette coefficient (≈ 0.51), which is a measure of how similar a point is to its own cluster compared to the other clusters. The *min_samples* parameter (the minimal number of neighbors a point needs to have within the radius to not be classified as an outlier) became a bit irrelevant because for the chosen *eps* the algorithm would always identify one cluster composed of 49 states. Only Illinois and Louisiana were classified as outliers.

This outcome supports the previous observation made from the DTW matrix. One of the possible reasons is the fluctuation of murder rates in these particular states. Both their time series are characterized by abrupt dips followed by high peaks that are typically not present in other states. Specifically, there is a stiff drop from 1984 to 1986 and again from 1987 to 1988 for Illinois while a similar situation can be observed also in Louisiana, especially from 1991 to 1992.

DBSCAN with Euclidean Distance

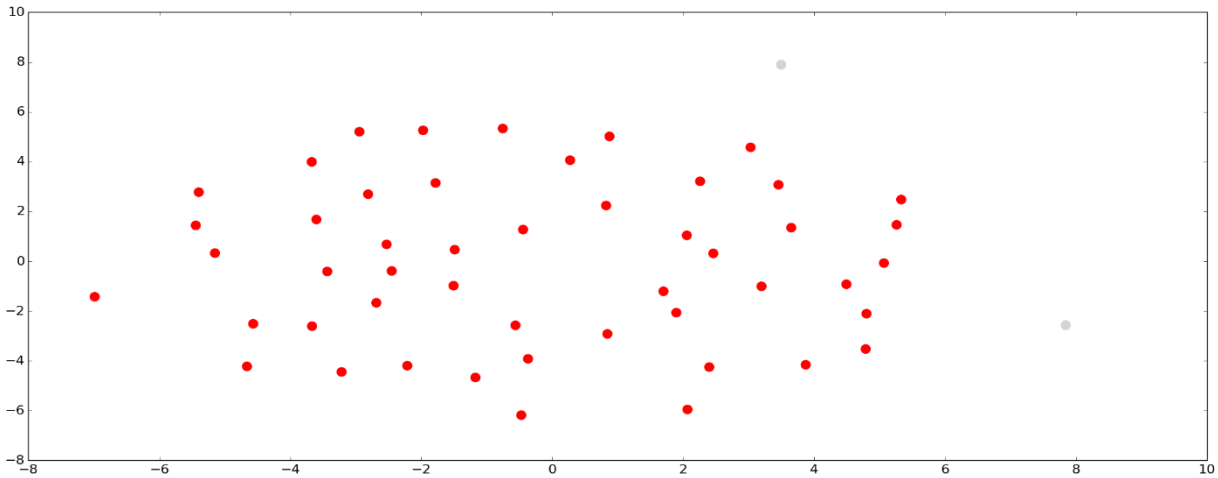
Now the DBSCAN algorithm classifies the points based on their Euclidean Distances. From a visual perspective, there is not a big difference between the results obtained with DTW as far as the distance matrix is concerned:

Figure 6 - Matrix representation of the Euclidean distances



The same procedure as the above was followed in order to find the best parameters for DBScan. For *eps*=7 and *min_samples* ranging anywhere between 2 and 39 the highest silhouette score of 0.43 was achieved. It should come as no surprise that again, the states of Illionois and Louisiana were the only ones labelled as outliers, while the rest were classified into a single cluster.

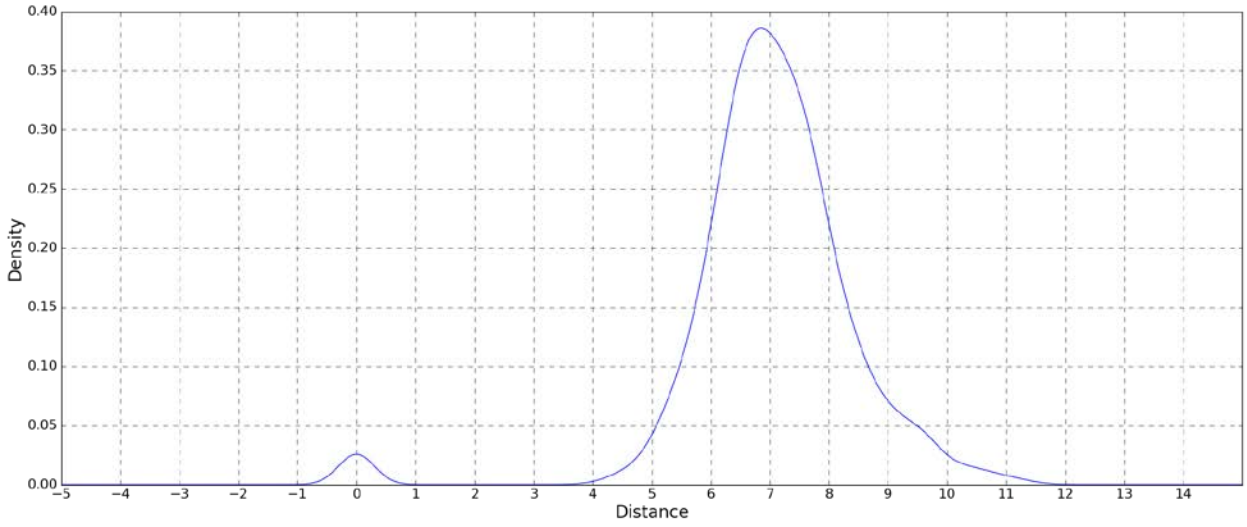
Figure 7 - Reduced dimensionality representation of clusters



Using sklearn's *manifold* library, above we plotted a reduced dimensionality representation of each state, based on the dissimilarity (distance) of their features. The light gray points depict the two outlier subjects while the only cluster found by the algorithm is composed of the red points.

The kernel density plot below estimates the probability density function of the distances between the states. The most common distance seems to be approximately seven units and the majority of the data is spread around this value:

Figure 8 - Kernel density plot of Euclidean distances



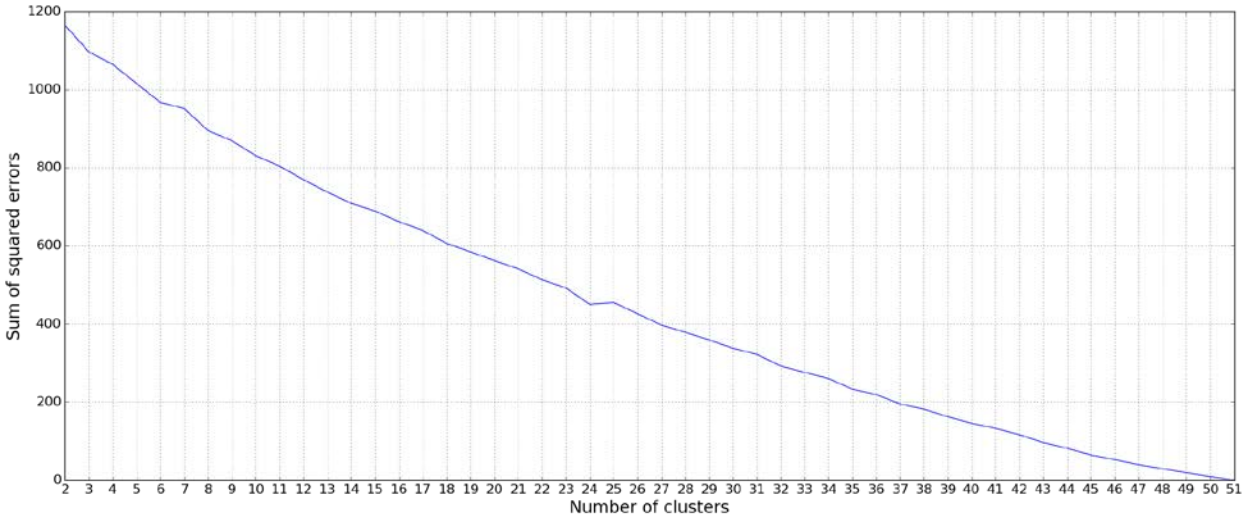
K-Means with Euclidean Distance

In the context of time series analysis, the normalized data used in the previous analyses will also be used for the K-Means algorithm without undergoing any further changes.

The ultimate goal of this algorithm is to minimize the sum of squared errors (SSE). In other words, the Euclidean distance of each data point to its closest centroid in calculated and then the total sum of squared errors can be computed. As the number of clusters k increases, the total SSE decreases because the number of points assigned to each centroid becomes smaller.

The problem with too many, too small clusters however is that they do not provide useful information for unsupervised discretization. Thus, a balance between a low SSE and adequate *k* must be found. For this purpose, the "elbow" technique can be used, i.e. we choose k such that if the number of clusters is increased by one, the contribution to decreasing the SSE will not be significant. Below we show the SSE graph:

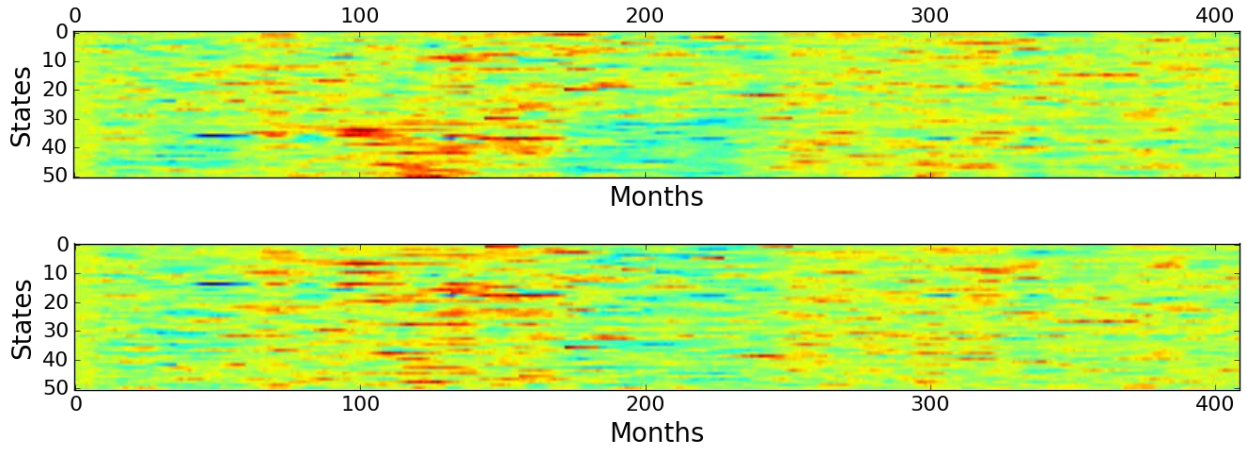
Figure 9 - SSE distribution



Unfortunately, in this instance, our elbow is rather weak so there is no clear point that we can choose to be the optimal number of clusters. Furthermore, we computed the silhouette scores for K-Means with the number of clusters ranging between 2 and 51 (the maximum possible number). The scores obtained were slightly above zero, the highest being ≈ 0.1 and suggesting only two clusters as the ideal number. By applying this parameter, we got one cluster containing 31 states and another with 20 states.

Using the freshly obtained labels, we grouped the states belonging to the same clusters and compared them with their original order via a matrix representation as show below (grouped subjects are on the top graph):

Figure 10 - Matrix representation of clustering differences



The visual difference between the two representations is not particularly noticeable. In the last step, we computed some of the features of the dataset such as the mean, standard deviation,

skewness and kurtosis. After applying K-Means on the new set, the cluster labels were mapped back to the states that had these specific features. However, the classification does not differ greatly from the previous assessment.

Conclusions

Grouping the dataset by month revealed some interesting patterns that were present across the majority of the states. The data indicated that more murders had in fact been committed during the months of July, August and December while February was generally less plagued by homicides. However, this apparent periodicity is not very strong. As the autocorrelation plot suggests, in most cases the generated "signal" is between the 95% confidence interval, meaning that we cannot claim a strong correlation within the series. The general trend is characterized by an increase in the murder rate in the late 80s and early 90s, followed by a slight decrease heading in the 2000s.

Applying DBSCAN on the DTW and Euclidean distances yielded very similar results. Forty-nine states were assigned to one big cluster while the two remaining ones were labelled as outliers, most likely due to their heavy fluctuations in the number of homicides.

K-Means produced a more balanced split between the two clusters it generated. States with more similar features were brought together, with the outcome however not being particularly revealing.

Classification

Introduction

The original Titanic dataset that you can find on Kaggle consists of 10 features related to every single passenger of the Titanic, giving us the personal information of a passenger such as his sex, age and information about his trip, such as his ticket class and port of embarkation. The target feature ('Survived'), instead, tells us whether a passenger survived the tragedy or not; so the task consists of predicting as accurately as possible what sorts of people were likely to survive.

Tools and Technologies Used

This project has been developed using Python 3.6 and its useful libraries "Pandas" (0.20.1), "Scikit-Learn" (0.18.2) and "Matplotlib" (2.0.2).

Preprocessing steps

During the initial analysis phase, it can observed that there are features, which are too specific and irrelevant to the task, such as the ticket number of a Passenger or his ID. Therefore, we have decided to remove these attributes; in the training set, there are also some missing values that we have handled using the operations made in the *ipynb* file supplied here.
In the end, we worked our way to the next classification phase using the following attributes:

- **"Survived" -** a Boolean which reports whether a passenger survived or not;
- **"Pclass",-** a categorical attribute that refers to the ticket class(1st, 2nd or 3rd) of a passenger;
- **"Fare" -** a numerical attribute indicating the price of a passenger ticket;
- **"Sex"**- categorical variable referring to the sex of a passenger;
- **"Embarked" –** categorical variable that discloses the port of embarkation of a passenger (it has 3 possible values: 'C', 'Q' and 'S')
- **"Age" -** discrete attribute that reveals the age of a passenger;
- **"FamilySize" -** discrete attribute that tells us the number of relatives embarked (it is obtained by summing of the original attributes 'SibSp' and 'Parch', for every passenger).

Classification

For the classification task, we have decided to use different features and different ways of building a model and then testing it. In particular, analyzing the "importances" of the features obtained by running multiple Random Forest Classifiers by using 'entropy' or 'gini' as split criterion and maxDepth between 2 and 15. In addition, the selected class weights were 'Balanced', 'None', or 0.3/0.7 for Not Survived/Survived respectively. By averaging on the features importance given by every single Random Forest Classifier, we concluded that the 'Embarked' attribute is a very poor indicator to predict the target value. We have built every model using four distinct settings:

	SETTING 1:	**SETTING 2:**	**SETTING 3:**	**SETTING 4:**
Features Used:	All	All	All except 'Embarked'	All except 'Embarked'
Features Normalization:	None	Z-Score	None	Z-Score

Then, for every model we exhibit the accuracies obtained testing it on three distinct test sets: the first one, made of 20% of examples of the original dataset; the 2nd one, made of 30% of examples; the 3rd one, made of 40% of examples. We will call them 'Test20', 'Test30', 'Test40' respectively. Finally, we show the accuracy obtained using tenfold cross validation. The confusion matrix shown, instead, will refer to the results obtained using 'Setting4' (that usually reaches the best results) on the 'Test30', because it is the most used in literature.

Naive Bayes

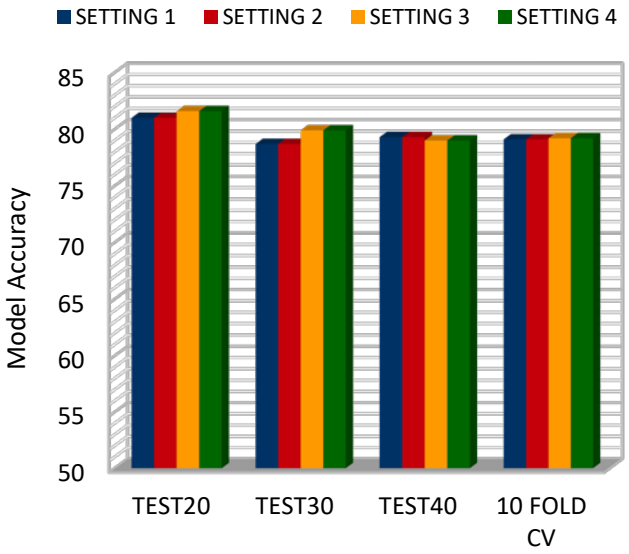
For the Naive Bayes classifier, using the three possible distinct models in scikit-learn (MultinomialNB, GaussianNB and BernoulliNB), we have tried to analyze the results obtained. Unfortunately, there are only few parameters that you can play with on these kind of models; in fact, they are very linked to the data they work on; so we have used the standard parameters (using Laplace parameter as smoothing parameter in the likelihood calculations). Trying the three distinct models on the four settings listed above, we have reached the best results using the Gaussian Naive Bayes model. What is very relevant to report about the different results obtained is the high independence of the models built with respect to the training set used. So, what's emerged is:

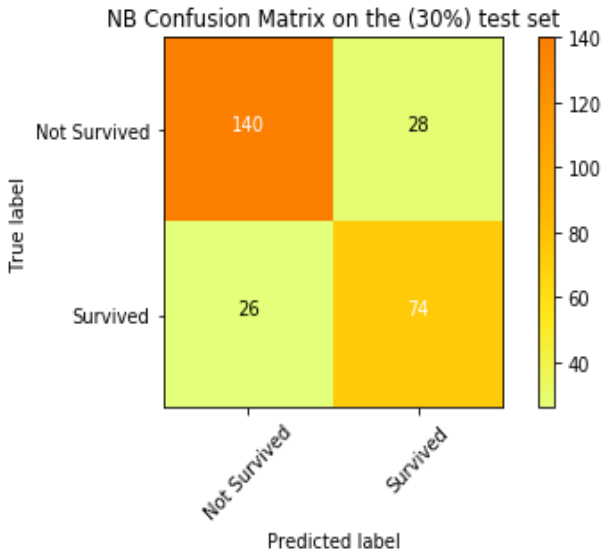
- Even with a relatively small training set, the performances of these kind of models on the test set were pretty good;
- The results achieved by the distinct models are pretty similar between themselves, even using relatively small training sets;
- The results obtained on the test sets are not so poor with respect to the results obtained testing the model on the training sets.

These very interesting points can be explained considering the simplicity of the Naive Bayes, which builds its own model by naturally self-tuning on the data in input. In particular, the fact that the performances are good enough even using a very small training set (even just the 10% of the original one). Even if it is not important in our task, this represents also a huge advantage in terms of efficiency: in fact, the classifier turns out to be accurate even if built on a small subset of instances, considerably speeding up the time needed to calculate the likelihoods and to predict new instances.

Finally, we show below the results obtained by the Gaussian Naive Bayes classifier, using the Laplace formula to calculate likelihoods of the features.

Figure 11 - Accuracies and Confusion Matrix for the Gaussian Naive Bayes classifier



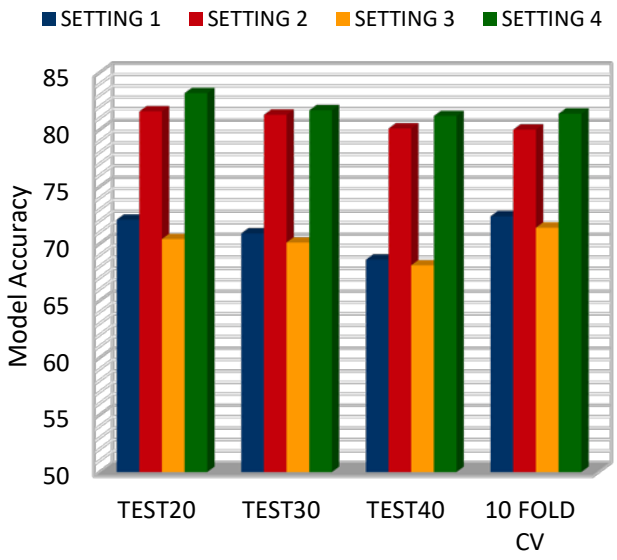


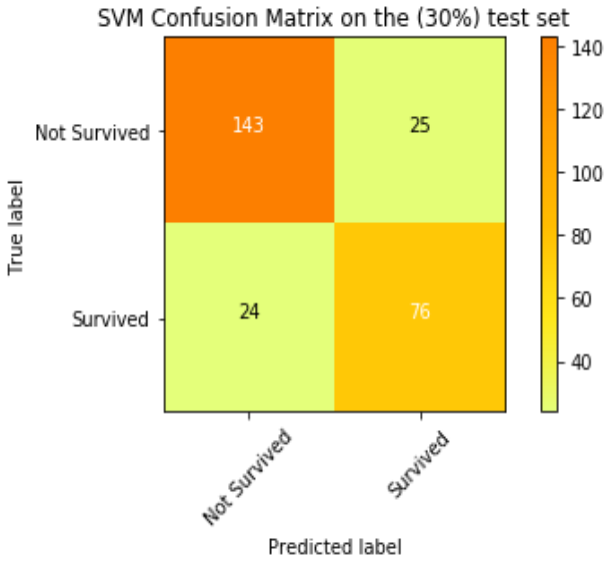
As you can see, removing the 'Embarked' feature (settings two and four) usually increases the accuracy. Instead, the normalization of the features seems to be completely irrelevant. In the end, a sort of "regularity" emerges between the distinct results obtained. Finally, from the confusion matrix we see that it gives a Precision score of 72.55% and a Recall score of 74% for 'Survived'.

Support Vector Machine

Also with SVM, we tried to adjust its parameters to reach better results. One very important parameter with this kind of model can be the kind of kernel chosen: we tried utilizing the linear kernel, the polynomial one (using a degree between 3 and 10), and the radial basic function one. The polynomial one slows down the building phase quite a bit, and does not improve the accuracy with respect to the other kinds of kernels. The linear one, instead, gives results very homogeneous even varying the features used; finally, the RDF one, is very sensitive to the normalization of the features. In fact, it was shown to improve the accuracy of the 10% in average just building the model on the normalized features instead of the original ones; it has so given the best results in these cases, and so we've chosen to report it in the next section(about the results obtained). Then, we gave different weights to the two classes: we saw that by increasing the weight for 'Survived', the recall also increased. Vice versa, increasing the weight for 'Not Survived', the precision obtained was much higher. Considering that, the task of the classification is not to save lives or to predict every single "death" of the tragedy with a high recall we decided to use a balanced weight for the two classes, obtaining also the best results in terms of accuracies. So, we show below the results obtained by a SVM trained using the following parameters: PenaltyParameter=5; Kernel=RBF; ClassWeight=balanced.

Figure 12 - Accuracies and Confusion Matrix for the Support Vector Machine classifier





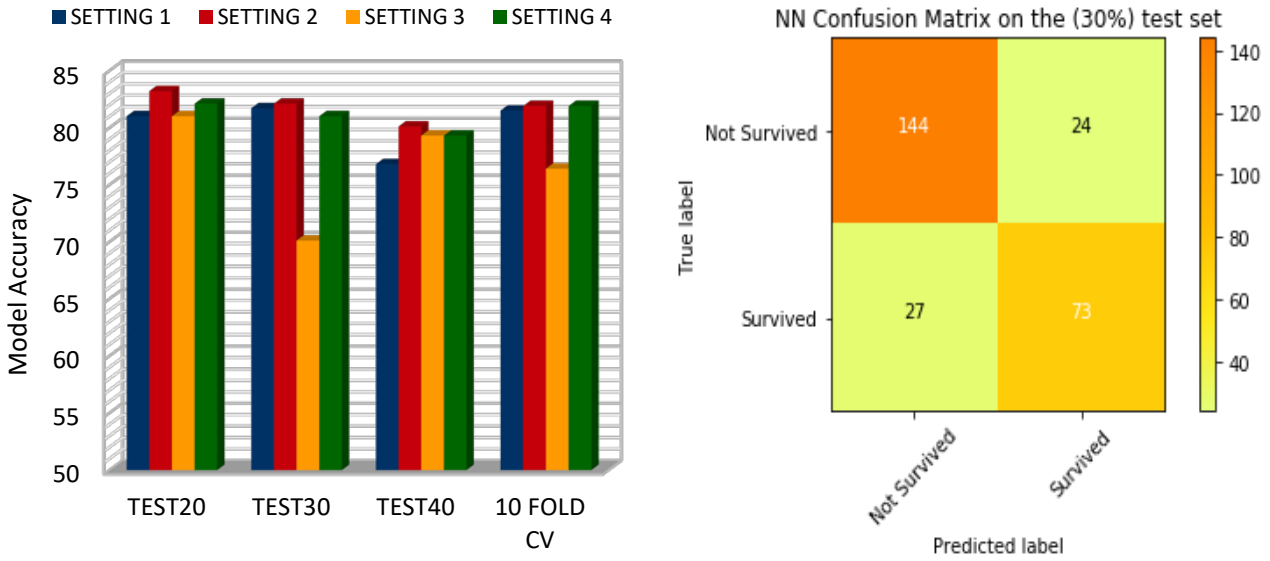
In this case, the normalization of the features literally rears up the accuracy; then, as you can see, the results on the test sets decrease (not drastically) with the increasing of the test set size. As with the NB classifier, the best results are obtained using normalized features and removing the 'Embarked' feature (setting 4). Here we get a Precision of 75.25% and a Recall of 76%. The Accuracy, instead, is 81.72%.

Neural Networks

A very important choice within the Neural Networks model is the number of the inner levels of the network, and the number of nodes in every level. Trying to use more than two levels, we obtained too specialized models that were unable to correctly predict the test set instances. Therefore, we have chosen to use two inner levels that have increased the accuracy a lot with respect to models with just one inner level, on training and test sets too. For what concerns the choice of the number of neutrons used in the two levels, we obtained the best results using six neutrons in the first level and three in the second one. Using more than six neutrons in the first level has yielded, again, a too specialized model, that has not been able to predict many instances of the test sets used; the same still holds by using more than three neutrons in the second level. Finally, these choices turned out to be quite good in order to avoid "overfitting" and to decrease the variance in the choice of the training/test sets used: using less than six/three neurons in the 1st/2nd level, the results obtained may vary quite a lot using distinct training sets. This inconvenience, together with the proper choice of the network levels, is smoothed a lot using normalized features, as will be shown in the next "Evaluation" section.
Finally, these results (in particular the relatively small number of the layers) are very likely linked to the relatively small size of the training set, which contains only 891 examples; the neural networks, in fact, typically improves its performance with an increasing number of examples to learn from.
Regarding the other parameters, the choice of the 'lbfgs' solver for weight optimization has been crucial: in fact, according to what is said on the Scikit-Learn official documentation: "it can converge

faster and perform better, for small datasets". In fact, using it has proven to improve the results considerably, with respect to other solvers, again showing the smallness of the dataset. Then, we tuned the parameter "alpha" (which refers to the "grade" of penalty), but none of the tried values seemed to improve the classification performances with respect to the standard one (0.0001). The results shown below have been obtained using the following parameters: hidden layers=2; 1stLevelNeutrons=6; 2ndLevelNeutrons=3; solver=lbfgs.

Figure 13 - Accuracies and Confusion Matrix for the Neural Networks classifier



Again, what emerges is the positive impact of the normalization on the accuracies. Here, with respect to the two previous models exhibited, the results obtained on the distinct test sets are not so "regular"; however, in every test set the best results are reached by using all the features, with normalization (setting 2). The Precision obtained using this model is 75.26%, the Recall is 73% and the Accuracy is 80.97%.

K-NN

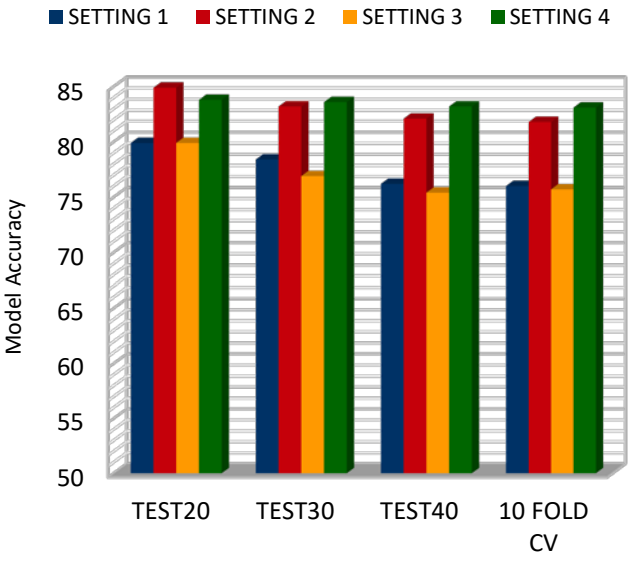
The built K-NN classifiers have been thoroughly analyzed in terms of parameters used. This kind of model is in fact highly susceptible to the parameters specified by the user, in particular to the number of neighbors chosen. It is easier to understand the results obtained and then trying to improve them tuning the parameters in a different way.
Using the KNN Model defined in "Scikit-Learn", we tried to improve the results obtained in terms of accuracy using the parameters listed below:

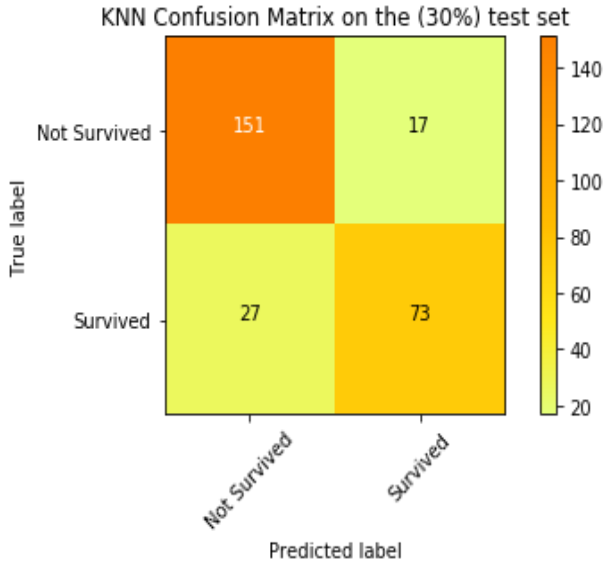
- **K** (the number of nearest neighbors used to predict the target value of a new instance), **between 2 and 50**;
- **the Minkowski metric** as distance metric to measure the similarity between two instances;
- **p** (that refers to the power parameter in the Minkowski metric formula) **between one and five;** of course, when it's set to one, the Minkowski metric corresponds to the Manhattan distance; when it's set to two, the metric corresponds to the Euclidean one instead;

- **uniform** (every neighbor has equal importance to decide the class of a new instance) **or distance based weights** (the more similar a neighbor is, more influential it is to "decide" the class of a new instance).

This way, we have generated every possible model using these parameters, analyzing the performances obtained and trying to explain the results: what we have obtained is very different with respect to the previous kinds of classifier analyzed. In fact, the generated KNNs have given very good results on the training sets, even with very low values for k, but these results worsened quite a lot on the test sets. Being that the results on the training sets are relatively uninteresting, we instead talk about the results on the test sets: the best results have been obtained using normalized features. It seems to increase the accuracy with about 5%, with respect to the original features; it is also interesting to highlight that the number of *k* needed to reach a good accuracy is relatively small in this case, contrary to what happens without normalization. The best values of *k* seem to be in the interval between 10 and 20, and in some particular case even between 5 and 10. Instead, using the original features(without normalization), what we've seen is that K needs to be relatively high to get better accuracy; this seems to happen in particular using relatively small training sets: maybe the model needs to learn more deeply, or the nearest neighbors features are not so discriminating. For what concerns the metrics, instead, usually the Manhattan distance seems to give the best results, but it does not seem to be as much influential. Instead, the weight is very significant for determining a better accuracy: in fact, using the normalized features, the 'uniform' weight gives the best results. On the other hand, using the original features seems to be non-influential; finally, what we want to point out is that it does not seem to depend on the number of neighbors used: in fact, we could expect that increasing the number of neighbors, the distance-based weight should perform better, but this does not happen.
Below, for every test set and every distinct setting, we report the best results in terms of accuracy obtained by KNN (the parameters used to build the models are not necessarily the same). The confusion matrix obtained on the test set, was generated using the model that has reached the best results on the Test30 (its parameters are K=11, distance_metric='Manhattan', weight='uniform').

Figure 14 - Accuracies and Confusion Matrix for the K Nearest Neighbors classifier



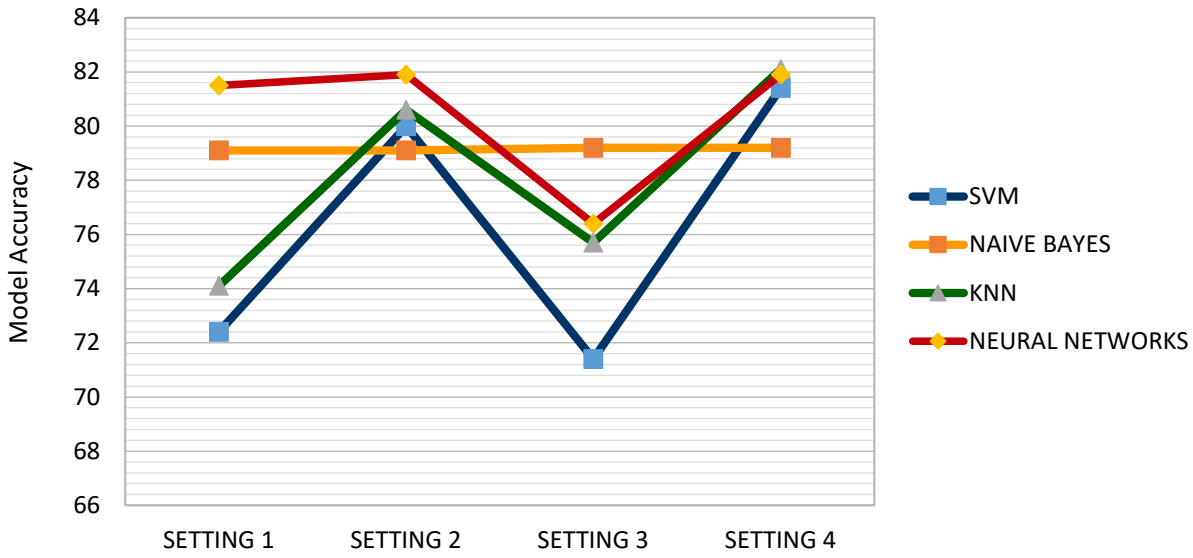


What we can see is that the normalization, again, improves the accuracy. In this case, the removal of 'Embarked' attribute decreases the accuracy of the model. Finally, we can see that the accuracies usually decrease by increasing the size of the test sets (i.e. by decreasing the size of the training sets), in particular for the settings 1 and 3. From the confusion matrix shown, we can see that we have reached a Precision score of 81.11% and a Recall score of 73% with the setting 4 on the 30% test set, with an Accuracy of 83.58%.

Conclusions

In the end, we show a graph representing the comparison of the accuracies obtained by the four distinct kinds of classifiers presented. For every kind of classifier, we show the results obtained using every setting; the results have been obtained using tenfold cross validation. For the KNN classifier, we have selected the best one, averaging the accuracy obtained using the four distinct settings. Its parameters are as follows: K=9, distance_metric='Manhattan', weights='uniform'.

Figure 15 - Classifiers accuracies comparison using tenfold cross validation



The Neural Network model gives the best results in the first two settings (keeping 'Embarked' in the features list), the third setting instead shows a huge decrease of the accuracy, with the only exception of the Naive Bayes classifier which, as mentioned earlier gives a pretty standard accuracy, independently from the instances settings used. As for setting 4, that in average gives the best results, the KNN performs better, but even SVM and K-NN give similar results in terms of accuracy. More or less with every setting, the SVM model seems to perform worse.

For what concerns the different confusion matrices obtained, we can conclude that every model performs better for 'Not Survived' in terms of Precision and Recall. This is probably caused by the unbalanced distribution of the two classes in the dataset.

Outlier Detection

Anomaly detection with DBSCAN

Given our set of two-dimensional points, DBSCAN will group together points that are closely packed together (points with many nearby neighbors), marking as outliers points that lie alone in low-density regions (whose nearest neighbors are too far away). By applying this algorithm with different combinations of *eps*, *min_samples* and *metric* parameters on the dataset, we found that there were several possible choices for yielding the same outcome: a 5% outlier fraction i.e. 15 points out of the total 300. Of course, the specific points classified as outliers varied depending on the values of these parameters. Below we show two scatter plots from the dataset with Euclidean and Manhattan distances used respectively:

Figure 16 - Outlier detection with DBSCAN – eps=0.123, min_samples=3, metric='euclidean'

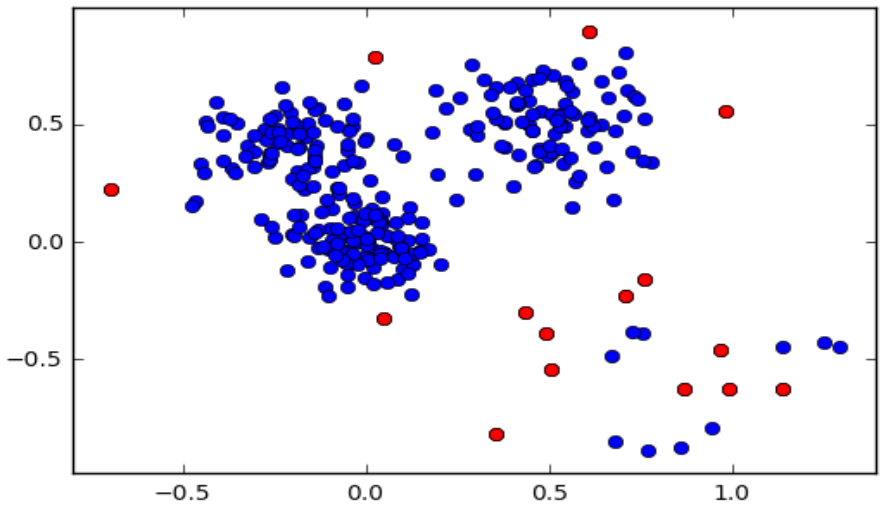
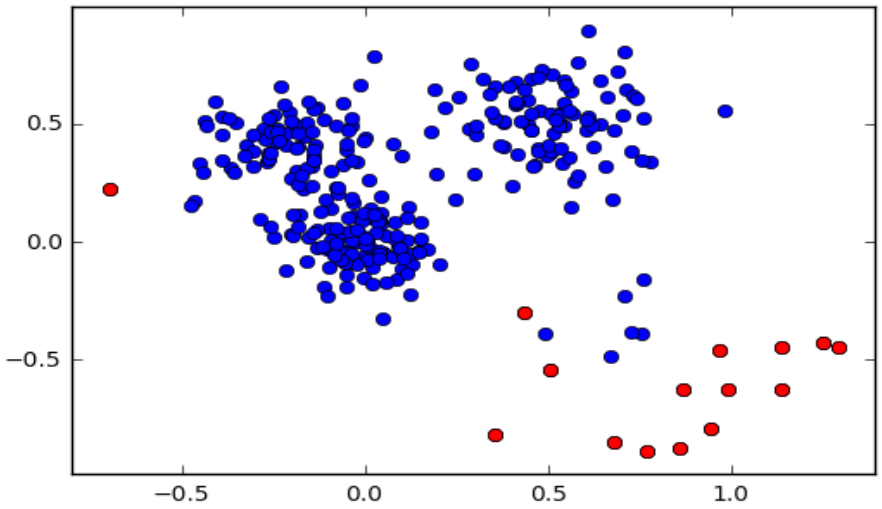


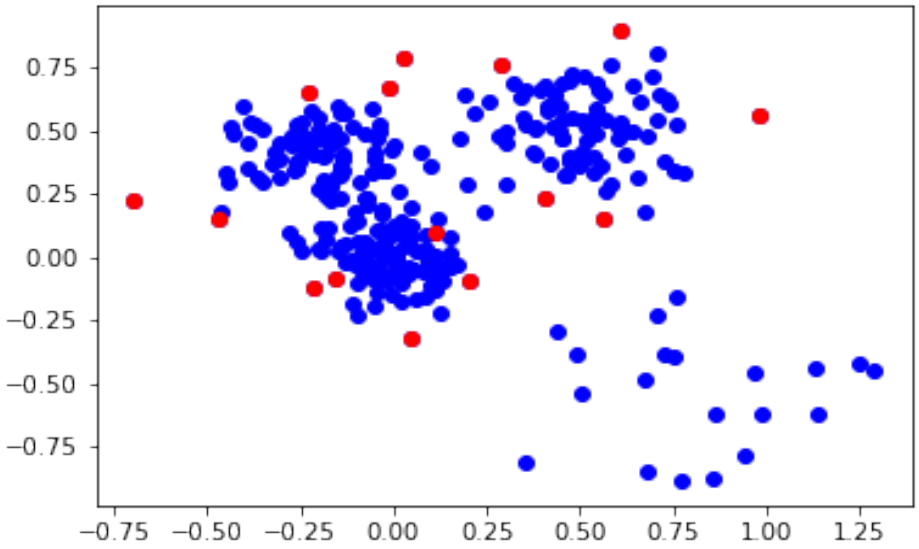
Figure 17 - Outlier detection with DBSCAN – eps=0.257, min_samples=6, metric='manhattan'



Anomaly detection with Local Outlier Factor (LOF)

LOF is based on a concept of local density. By comparing the local density of a point to the local densities of its neighbors, regions of similar densities can be identified and points that have a substantially lower density than other neighbors are classified as outliers (usually for LOF > 1). Finding a usable LOF library for Python was a little tricky, since the official version from *scikit-learn* is not available yet. For this, we downloaded the whole *scikit-learn* project from Github and then copied the "lof.py" file available there into our local library to make it accessible. We chose the same number of nearest neighbors and metric as with DBSCAN in order to highlight the differences in the results obtained:

Figure 18 - Outlier detection with LOF – n_neighbors=6, metric='manhattan'



Judging by the points that LOF chose to label as outliers, there is a notable distinction from DBSCAN's classification. The outcome might even be viewed as a bit unintuitive. This is because LOF takes into account the reachability distance of each point from its nearest neighbors. The lower the LRD of a point, the further away it is from its neighbors and the higher is its LOF coefficient.

Conclusions

Although DBSCAN and LOF share similar concepts such as "core distance" and "reachability distance", in some cases they may achieve different results even with similar parameters, due to the contrasting philosophies in calculating the density of a neighborhood.