University of Pisa
Graduate Programme in Computer Science

Data Mining Project: Tennis Matches

Edoardo Federici, Michele Morisco, Carlo Tosoni

Data Mining
Academic year 2021/2022

1 Introduction

The project consists in the data analysis of a given dataset using data mining tools. The project was developed using Python 3.8.
The project consists of four parts:
1) Data understanding and preparation
2) Clustering analysis
3) Predictive Analysis
4) Time series analysis

2 Data Understanding

2.1 Studying the dataset

The dataset we received contains information about tennis matches split in three different .csv files:

- tennis_matches.csv where each row represents a match
- male_players.csv which contains information about male players
- female_players.csv which contains information about female players

The dataset contains many different features, including **'tourney_id'** that is an identifier for each tournament. Every tournament is subdivided into matches that have the same id. For each match we have information about the winner and the loser's player name, stats of the match like the number of aces, the number of service points, the number of doubles faults, and so on.
The dataset also keeps track of some information about players such as height, age, and nationality. Finally, each tournament present in the dataset was played between 2016 and 2021.

2.2 Data Quality

A quick analysis showed that the dataset was ripe of missing values.
We decided to deal with duplicated records by removing them altogether since we subsequently devoted a great effort in trying to save as many missing values as possible cross referencing the possible deductible attributes. For example, we tried to reconstruct the **'tourney_id'** attribute following as closely as possible the original structure, or as we did for the **surface**, where we noticed that we could look at other attributes to unequivocally determine it.
We also found unacceptable records in the **Name** attribute, hence we decided to remove them.

2.2.1 Missing values

In the original general dataframe there were many missing values on various attributes. We decided to treat them in different ways; below are listed some of the attributes of the file *'tennis_matches.csv'* that we dealt to develop our project.

- **tourney_id**: we used the date, the level and city of a match to understand which of the matches that had a null value on that attribute could have been played in the same tournament. Hence we used that information to replace these missing values.
- **surface**, **draw_size**, **tourney_level**, **tourney_spectators**, **tourney_revenue**: Since they are the same in each tournament, we used the values in the matches played in the same tournament to replace these missing values. The missing values remained were corrected using the most common value in the dataframe.
- **winner_id**, **winner_name**, **loser_id**, **loser_name**: Assuming that there aren't cases of homonyms in the dataframe, we replaced the missing values in these attributes using the information stored in the records having the same id or name (in both winner and loser attributes).
- **loser_hand**, **winner_hand**: In these attributes there were also a lot of 'U' (unknown) records in addition to the missing values. Assuming that players usually play with the same hand, we replaced them using values contained in records about matches were the same player played.
- **loser_ht**, **winner_ht**: We treated them in a similar way to **loser_hand** and **winner_hand**.

- **winner_entry**, **loser_entry**: Since the entry of a player does not change for the duration of the tournament, to replace the missing values we looked at those records where the same player played in the same tournament.

Moreover, about 15 records were deleted since they contained too many missing values, and therefore they were not reliable.
The attributes about the single matches (like **minutes** or **w_ace**) were not touched; firstly because we could not obtain that information from other records and secondly because there were too many missing values in those attributes to replace them using the average or the median.

2.2.2 Outliers

In this section, we studied the dataset's outliers comparing the distribution before and after their removal.

2.2.3 Duration of the match

We started the study of the outliers analyzing the matches' duration, because we noticed that some matches were too short or too long. On the dataset, this information is represented by the **'minutes'** attribute. For detecting the outliers we used the Interquartile Range (IQR) proximity rule.
Firstly, we got the 25th (Q1) and 75th (Q3) percentile of the 'minutes' values on the dataset. After that, according to this rule, the data points which fall below Q1 – 1.5 IQR or above Q3 + 1.5 IQR are considered outliers. The IQR represents the inter-quartile range and is given by Q3 – Q1.
The *figure 1* shows the distribution of the **'minutes'** attribute; we expected these values to be approximately normally distributed, but the chart clearly shows that some of them are wrong because it cannot be possible that there are matches longer than 4000 minutes. The IQR proximity rule lets us detect the upper and lower limit and the *figure 2* shows the values normally distributed as we expected.
Even after that we had weird values on the lower limit (we saw that some matches lasted just 30 minutes or less) hence we realized that these values were wrong. To remove these outliers, we searched the correct information on the wiki and we used it to correct these values.

After this analysis we understood that in this attribute there were still possibly wrong values; we searched the duration of the longest and the shortest matches ever played and we used that information to correct our data.

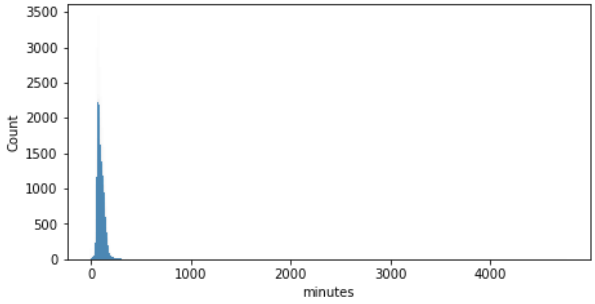

Figure 1: Distribution on 'minutes' attribute before capping.

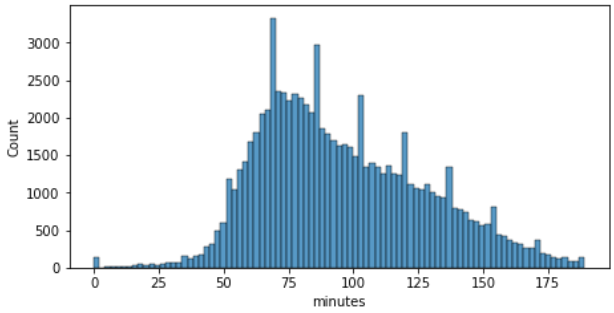

Figure 2: Distribution on 'minutes' attribute after capping.

2.2.4 Data correlation

In this section, we studied the correlation between spectators and revenue based on the attribute **tourney_size**, using the Kendall correlation.
We wanted to prove that the number of spectators and revenues of a tourney were directly proportional to the tourney size represented by the **'draw_size'** attribute assuming that the bigger is the tournmanet the bigger is the **'draw_size'**. In *figure 3*, the scatter matrices show the correlation between these attributes.
We have also studied the correlation between the number of first service points between winners and losers, showing the results for both the right-handed and left-handed players. This analysis proves that there is a close correlation as we expected, and *figure 4* shows the result with a scatter plot.

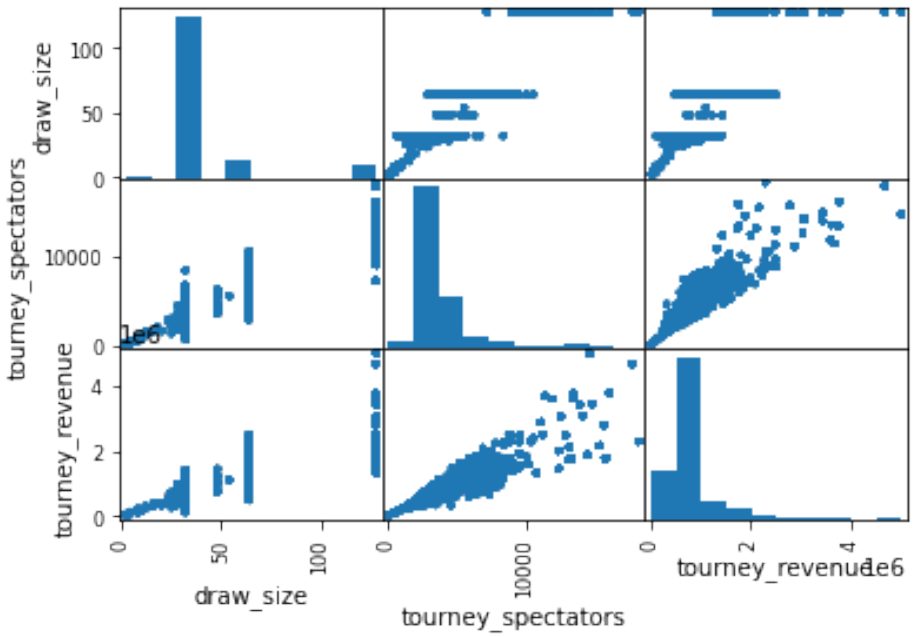

Figure 3: Correlation between number of spectators, revenues and tourney size.

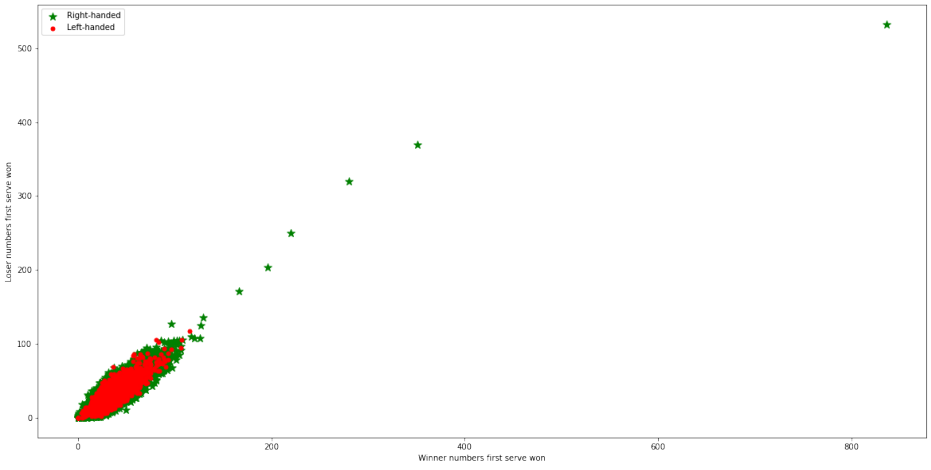

Figure 4: Correlation between numbers of first serve point between winner and loser players.

2.3 Data distribution

To study the distribution of our data and to highlight some particular features about it we created new dataframes; these are:
df_player_data: Includes information about the performance of players in matches. Every record of the original dataframe is split in two records: one for the loser and one for the winner of the match.
df_player_data_nd: Contains information about the participation of the players in a tournament. It has

one record for every participant in each tournament.
df_tourney: Stores information about tournaments, like the revenue and the number of spectators. It contains one record for every tournament.

In the notebook we looked at the distribution of many attributes, here we report just an example.
In particular we studied how points among players are distributed and we noticed that the average female player has a better score than the average male player, even though the players with the highest score are, more or less, all males. In *figure 5*, we can observe that behavior.

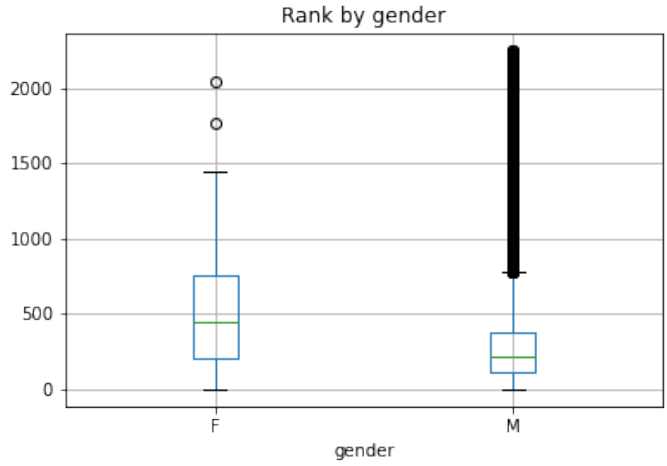

Figure 5: distribution of points by gender.

2.4 Merging the dataframes

To work on a single comprehensive dataframe we had to merge the three input files. We checked for missing values both on the **Female** and on the **Male** datasets, we combined the **Name** and **Surname** attributes in a unique attribute called **full_name** and then we added another attribute for the **Gender**.
Finally, we merged the three datasets in a single one, using the **full_name** attribute as the pivot.

3 Data Preparation

3.1 Indicators

We created a new dataframe containing the performance indicators of the players. We called it **df_player_profile**; all its attributes and their eventual mathematical formula are listed below.
player_id, **player_name**, **player_ht**, **player_hand**: These attributes were already present in the original dataframe and they were retrieved directly from it.
total_wins, **total_losses**, **total_games**: The total number of victories, defeats and matches played by the player. We have also created new attributes indicating only the performance relative to each year between 2016 and 2021 (**16/17_wins**, **16/17_losses** and so on).
total_winrate: We used the attributes **total_wins** and **total_losses** to compute this attribute, which represents the percentage of victories. Again, we have also created other attributes about the percentage of victories in a specific year (**16/17_winrate**). $(total\_wins/(total\_wins + total\_losses))$
ace, **svpt**, **firstwon**, **secondwon**, **df**, **bp_s**, **bp_f**: They represent, respectively, the total number of aces, serve points, first-serve points, second-serve points, doubles faults, breakpoints saved and breakpoints faced performed by the player.
ace_per_svpt: Total number of aces performed per serve point. $(ace/svpt)$
first_serv_rt: Percentage of points scored in the first service. $(firstwon/(firstwon + secondwon))$
avg_df: Average number of doubles faults performed by a player in a match. $(avg\_df/matches\_played)$
avg_ace: Average number of aces performed by a player in a match. $(df/matches\_played)$
fvrt_level: Level at which a player played the most.
mc_entry: Most common entry of a player.
last_date: Date referring to the last time a player played in a tournament.
01/09/2021_age: Age of the player on 1st September 2021.
perc_bs: Percentage of breakpoints saved. $(bp\_s/(bp\_s + bp\_f))$

avg_minutes: Average duration of the matches played by a player.
highest_scr, **avg_scr**: Highest score ever reached by the player and average score of the player during their career.
SvGms: It represents the average number of serve games performed by the player.

3.2 Correlation and Distribution

Before applying the clustering algorithms, we deleted the most correlated attributes in **df_player_profile**. We discovered that some of them were highly correlated; some pairs of attributes, like **bp_s** and **bp_f** had a correlation higher than 0.99. Hence, we deleted some of them, and at the end, only 9 attributes remained, as we can see in *figure 6*, these attributes are not strongly correlated.

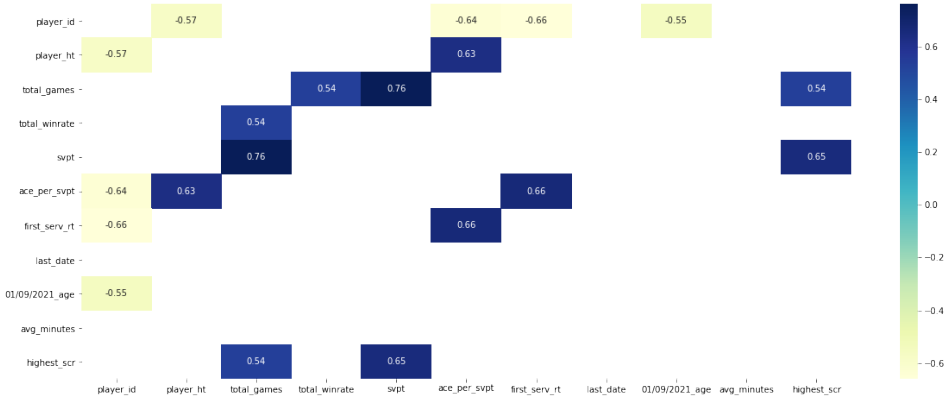

Figure 6: Correlation of the attributes after deleting some attributes for the clustering analysis.

After that we studied briefly the distribution of the new attributes, in order to understand which are the most typical behaviors of the players.

4 Clustering

Before studying the clustering, we have considered some attributes to normalize. After looking at our dataset we have decided to normalize the following attributes: **total_games"**, **"total_winrate"**, **"svpt"**, **"ace_per_svpt"**, **"first_serv_rt"**, **"avg_minutes"**, **"highest_scr"**.

4.1 K-means

One of the clustering algorithms that we used was KMeans. We ran it with a range of k spanning from 2 to 30 and for each of them we used 12 different random initialization, collecting the Silhouette Score and the Sum of Squared Errors.
We plotted both values with respect to k to find some candidate configurations (elbow method) which we then further analyzed by comparing the average Silhouette Score with the cluster size.
Finally for the selected k we plotted a radar chart and for all combinations of attributes we computed the resulting clustering labels and their centers.
The execution of the whole clustering pipeline showed that the best results were obtained with a k of 6, closely followed by a k of 4.
As we can see in *figure 7*, we have one dominant cluster (in this case labeled with 6) and many smaller ones. The same behaviour showed with all other tried ks, except for 4 where the clusters where most balanced.
All in all, both the SSH and Silhouette Score (for k=4,6) where in acceptable ranges.

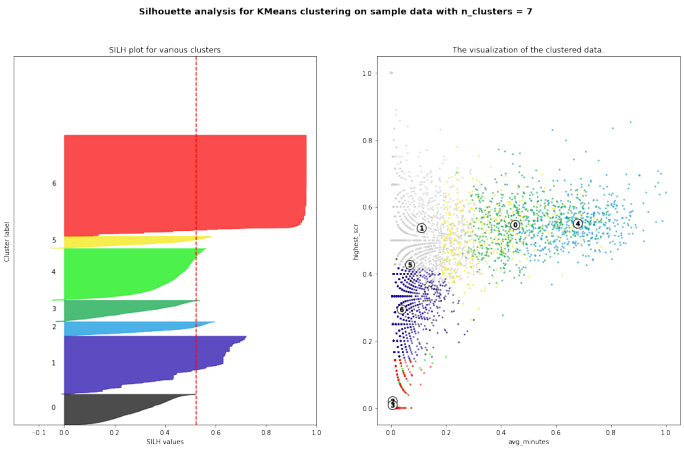

Figure 7: Silhouette analysis on the left, centroids and clusters over the first two attributes on the right.

4.2 DBSCAN

We applied the data clustering algorithm called DBSCAN on our dataframe. DBSCAN has two inputs: **eps**, which is the radius, and **min_samples**, which is the minimum number of neighbours required to be a core point.
In order to find the best values for the algorithm, we studied the sorted distances between the k-th neighbour and the dataframe's points. To compute this step, we used different values of k; from 2 up to 9, then, for every k, we plotted the graph of the distances in order to understand which was the best value of **eps**; to find that value we used the elbow method as shown in *figure 8.*

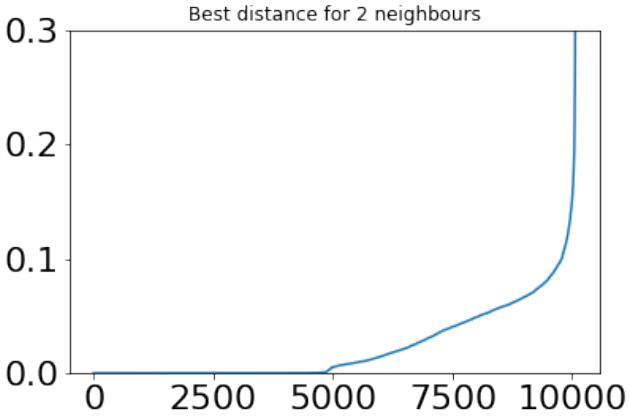

Figure 8: Graph of the sorted distances between the second neighbour and the dataframe's points. According to the elbow method, for **min_samples**=2 we can use **eps**=0.59.

The results of DBSCAN were similar for every value of k used, indeed the algorithm has always found three main clusters: one with about 5500 points, and other two having about 1200 points each. Moreover for every k there were about 1000 noise points.

4.3 Hierarchical clustering

For hierarchical clustering, our approach was to compare different merging strategies.
In our analysis, we have considered the following methods: ward, single, complete, centroid and average. First of all, we compared the dendrograms resulting from each method to find the most correct number of clusters; in this phase, we set a threshold to define which were the clusters. We have seen that the **single** method did not give us a good result, *figure 9* shows it was not able to properly identify the clusters. On the other hand, the **ward** method using a threshold equal to 10, *figure 9*, defined better the clusters than the single method.

Finally, we have noticed that the complete method shows a good enough result but still poorer than the **centroid** and **average** methods, as shown in *figure 9*.
After that compare the dendrograms, we have compared the clustering results using the dendrograms' results about the number of clusters. In this part, we have considered the following linkage criterion to use for agglomerative clustering:

- ward, that minimizes the variance of the clusters being merged
- average, uses the average of the distances of each observation of the two sets
- complete, uses the maximum distances between all observations of the two sets
- single, uses the minimum of the distances between all observations of the two sets.

Our results confirm that the single method is very bad as shown in *figure 10*, indeed, the plot shows only one big cluster. The other graphs show a better distribution of data defining more or less the clusters in a proper way. The better result obtained is the **ward** and **complete** methods: *figure 10* shows that **ward** method is better compared to the **complete** method.

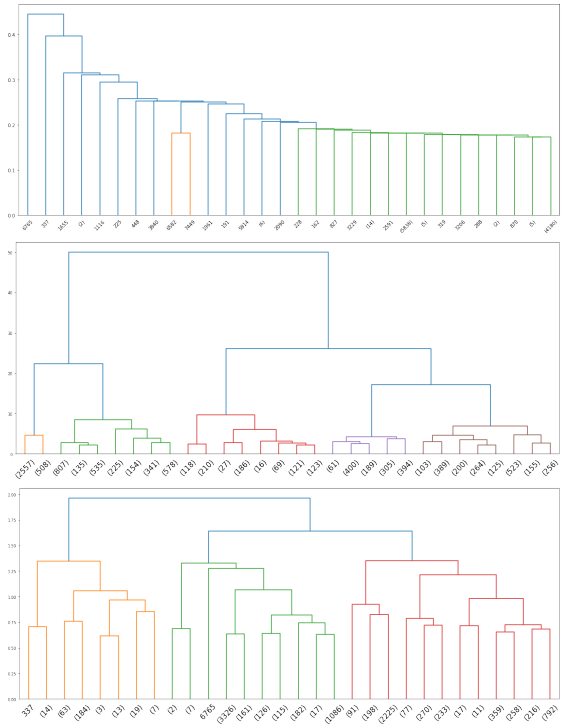

Figure 9: Dendrograms with single, ward and complete methods.

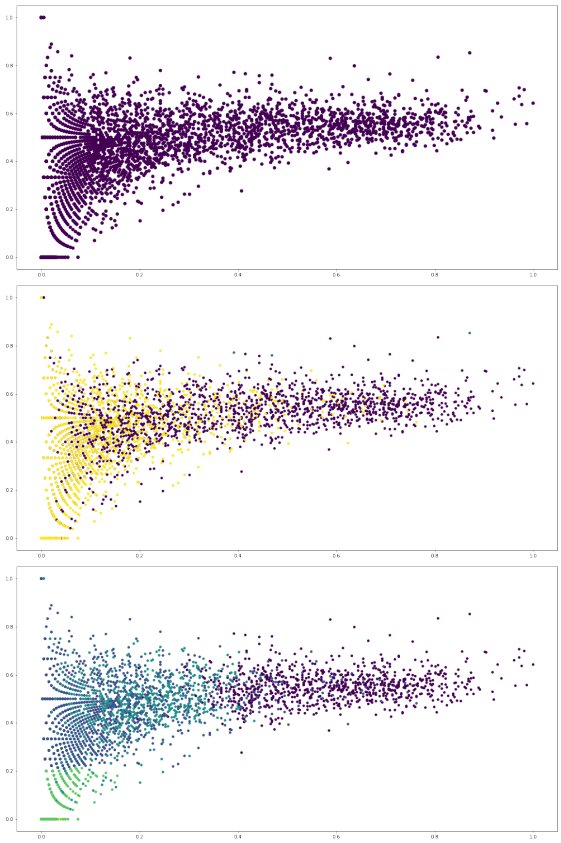

Figure 10: Results of hierarchical clustering with single, average and ward methods.

4.4 OPTICS

We chose **OPTICS** as our fourth algorithm for clustering. **OPTICS** is similar to **DBSCAN**, but in this algorithm the density is not constant, in fact it changes based on how the dataset's points are distributed in the dataset. Moreover **OPTICS** uses the two parameters of **DBSCAN** (which are **eps** and **min_samples**), but it adds two more variables called **Core distance** and **Reachability Distance**.

Core Distance: it is the minimum radius required to classify a point as a **Core Point**. The **Core Distance** is undefined if a point is not a **Core point**.

Reachability Distance: it is defined with respect to another point **q**. Hence the **Reachability Distance** is equal to the maximum between the **Core Distance** of a point and its euclidean distance with **q**. The **Reachability Distance** is not defined if **q** is not a **Core Point**.

The algorithm then plots the **Reachability Distances** of the points in a dendrogram in order to find which groups of points have the lowest distances among them.

The library that we used to implement **OPTICS**, i.e. **pyclustering**, requires also an additional parameter called **amount_clusters**, because the algorithm is not able to understand which is the optimal number of clusters by its own.

A problem regarding our dataset was that there are many points that had a lot of zero values on many attributes (these points represent the players that have never played in competitive tournaments or that have played only 1-2 matches). **OPTICS** is not able to cluster correctly these points, because they have a **Reachability Distance** equal or close to zero, that creates problems in finding the correct clusters by cutting the dendrogram of the **Reachability Distances**.

To solve this problem we created another dataset where these points have been eliminated. Doing that, we discarded about two thirds of the points.

Firstly, we executed **OPTICS** on this reduced dataset using some random parameters (**eps** = 0.1, **min_samples** = 5, **amount_clusters** = 3), below there are the results that we obtained, as we can see these results are unacceptable, the algorithm found only one huge clusters, because it was not able to exploit the different densities in the dataset.

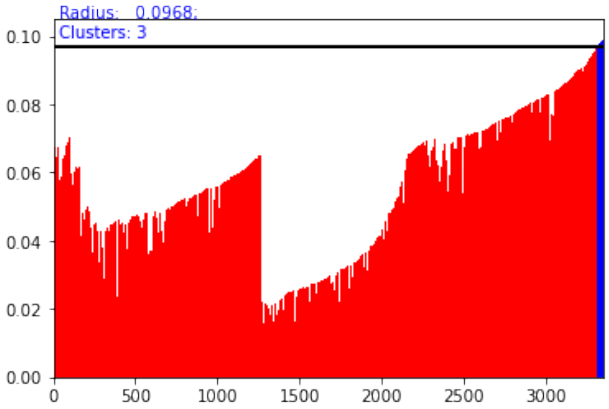

Figure 11: Dendrogram of the **Reachability Distances** (on the Y axis the **Reachability Distances**, on the X axis the points of the dataset)

We tried to improve our results by changing the parameters. We tried many possible combinations in order to see how the results changed. At the end we found a good set of parameters (**eps** = 0.065, **min_sample** = 5, **amount_clusters = 3**). Below there are the results that we obtained with these parameters.

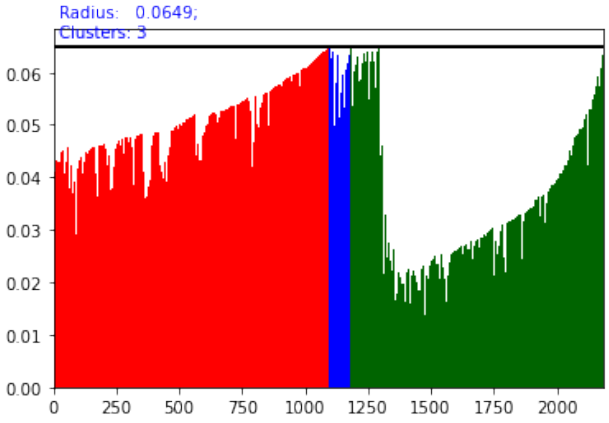

Figure 12: Dendrogram of the **Reachability Distances** with the new set of parameters

However, even in this case the results were not amazing, indeed with this set the algorithm found only about 2000 **Core Points** (In the original dataset there were about 10.000 points). Moreover, probably in our dataset we cannot divide the points based on the density, because even with this new set of parameters the results were quite poor.

4.5 Conclusions

The result of the two first algorithms, K-means and DBSCAN, were similar, even though there were some differences. Indeed K-means is not able to define clusters with a shape different to the globular one, while DBSCAN does it; on the other hand DBSCAN created also a great amount of noise points (about one-tenth of the total), while K-means did not leave points outside the clusters. However, at the end the best cluster algorithm was the hierarchical clustering with the ward method, indeed that algorithm was able to create better defined clusters than **K-means**, **DBSCAN** and **OPTICS**.

We have to keep in mind that we didn't aggressively change some missing values in the first part (like in the attributes **ace** and **svpt**) since we thought that they were recorded in the most part for the tournaments of the highest level and we didn't want to skew their distribution assuming the same behaviour for smaller tournaments. This resulted in many records with a 0 value on these attributes which surely changed the clustering algorithms dynamics: we think that they were aggregated together in the same cluster, resulting in a simpler representation of the real situation.

Moreover, we give an interpretation to the clusters created by the ward method of the hierarchical clustering framework. The 6 clusters identified by the ward method have clear differences that let us highlight why they were clustered together in the first place: bad players belong to clusters 2 and 4, with the only difference being that many players in cluster 2 didn't even have some stats recorded (we can infer that this was caused by the low level of the tournament). Clusters 1 and 3 are similar too and present the same difference as before: we can appreciate it even better because there is a greater spread between the average score of the players (indicating that players in cluster 1 are indeed worse than players in cluster 3). Finally, we can look at clusters 4 and 6, which clearly hold the better players; in cluster 4 we can find good players, with a high number of games played and good stats. Cluster 6 instead is reserved for the champions, with an astoundingly high number of games and a stellar average score.

5 Predictive Analysis

This task's goal is to classify the players according to their skill, assigning to each one of them a label, exploiting features from the original dataset and others previously computed.
The related subtasks are the definition of a player profile that facilitates the predictive analysis and the computation of the label for each player; we describe them in detail in the next section.
Finally, once we have the updated profile and a label for each player, we will use different models to predict which label we should assign to each player; we'll describe the techniques used in the following sections and we'll report our conclusions at the end.

5.1 Preprocessing

We started the prediction analysis by integer-coding the **fvrt_level**, **player_hand** and **mc_entry** attributes.
The next step we performed was the label computation. We identified 5 attributes to use for it: **first_serv_rt**, **fvrt_level**, **perc_bs**, **highest_scr** and **avg_scr**.
We filtered out of the 'high ranked' (1) label all the players that either most frequently play in low level tournaments or have a low score; then we refined the search selecting only players with certain stats (**first_serv_rt** and **perc_bs**) above a threshold; we decided those two stats as they've been proven to be highly correlated to the player skill according to various tennis studies (mostly here, but also here and here).
Since in the previous section we computed some derived attributes (the performance indicators), we analyzed the correlation between the various attributes of the dataset to filter away those that would erroneously impact the prediction (after removing the 5 attributes used for the prediction itself).
This process left us with 14 attributes to work with **16/17_wins**, **17/18_wins**, **18/19_wins**, **19/20_wins**, **20/21_wins**, **21/22_wins**, **ace**, **svpt**, **ace_per_svpt**, **avg_df**, **01/09/2021_age**, **SvGms**, **player_hand_num** and **mc_entry_num**.

5.2 Decision tree

First, we have used the decision tree for the classification task. We have tested two versions of the decision tree: one with the **'random'** strategy splitter and one with the **'best'** strategy splitter. In most cases, the **'best'** splitter strategy achieved good results even if there wasn't a great difference between them.
We have tested the decision tree with some parameters such as the *minimum samples split* and *minimum samples leaf*, achieving the best result with *4* and *6* respectively.
Figure 13 shows the decision tree that we have obtained with these features. The accuracy of the train and test set for this decision tree is quite similar, **0.98** and **0.96** respectively.
We noticed that the precision is smaller on the high-ranked label compared to the low-ranked players, this means that the model recognizes with less accuracy the high-ranked players.

Figure 14 shows the confusion matrix of the model where we have achieved the best results. We can see a lower recall on the 'high ranked' label. As we'll see later, this isn't an uncommon result for our analysis. Despite that, the results are pretty good: the decision trees performed very well in the players classification.

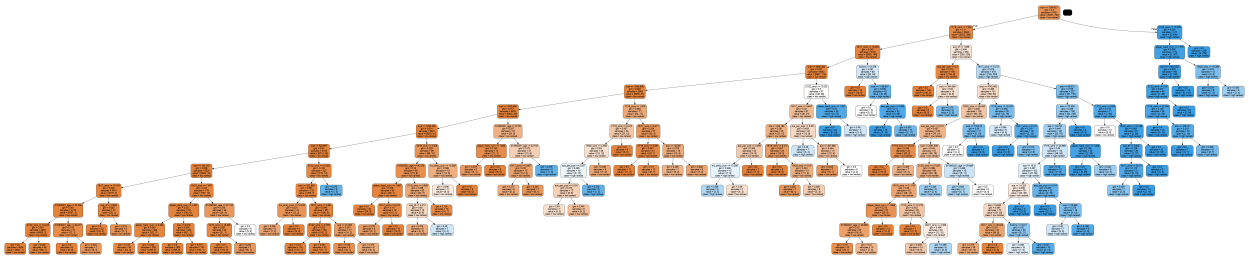

Figure 13: Decision tree with 'best' splitter strategy.

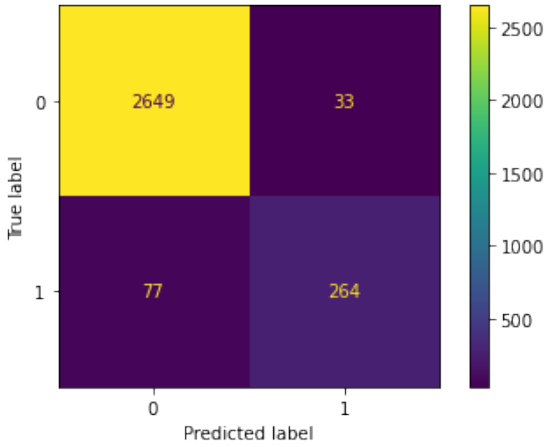

Figure 14: A result of confusion matrix with decision tree 'best' splitter strategy

5.3 Support Vector Machine

We tried to classify the points of our dataset using a **Support Vector Machine**. Usually, the **Support Vector Machine** is more precise when the points to classify are not many and the data dimensionality is very high. In our case, the dataset consists of about 10.000 records, which are not so many, but nevertheless, the **Support Vector Machine** could have some problems in finding a correct hyperplane to divide these points (since it works better with dataset of fewer points). Moreover, we are classifying the points using 14 attributes, hence even the data dimensionality is not very high.

Firstly, we tried to apply the **Support Vector Machine** on the whole dataset, searching for the best hyperparameters. We investigated the best values of **C** and **gamma** through a **Grid search**, using 4 values for **C** (which trades off misclassification of training examples against simplicity of the decision surface) and 5 values for **gamma** (which defines how much influence a single training example has; the larger gamma is, the closer other examples must be to be affected). We tried also to use a non-linear kernel using the **Radial Basis Function**.
The **Grid Search** found that the best parameters were **C** equal to **100** and **gamma** equal to **0.01**.
The results of this algorithm were really good; this method achieved a really high recall (0.99 on both the labels) and a slightly lower precision (only on the 'high ranked' players).
We tried to improve these results downsampling the points of the dataset. We created another dataset of only 2.500 points downsampling the original dataset, taking 500 with a **label** equal to 1 and 2000 with a **label** equal to 0. We trained and optimized this new SVM on the downsampled dataset and then we plugged into it all the points that the downsampling left behind. This yielded us almost the same results for the 0 label, but we overshoot the false positives of the 'high ranked' players by a large margin, plummeting its recall into 0.63.

Principal Component Analysis

We used the **Principal Component Analysis** to try to further improve the results of our **Support Vector Machine**. We can use the **PCA** to combine all the features of our dataframe into only 2 axes.

However, the results in this case didn't improve, indeed we achieved an accuracy slightly lower than the first model (the one without any downsampling strategy), probably because the approximation of the **PCA** was not suitable for our dataset.

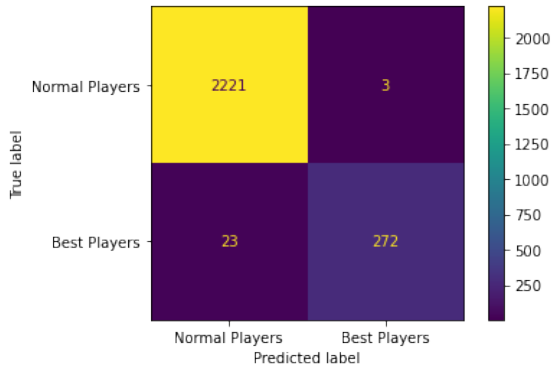

Figure 15: The confusion matrix of the first **Support Vector Machine** (without downsampling).

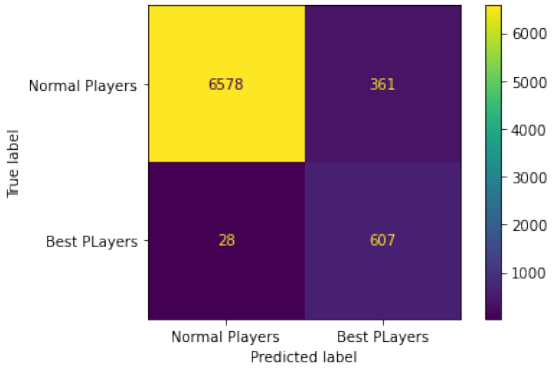

Figure 16: The confusion matrix of **Support Vector Machine**, after the downsampling.

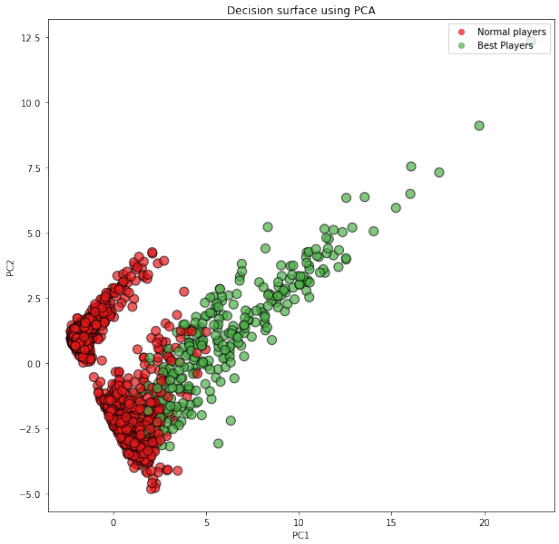

Figure 17: Result of the **Support Vector Machine** using the **Principal Component Analysis**

5.4 Neural Network

We decided to leverage the power of neural networks to perform the prediction analysis.
Given the custom function we implemented to assign the label to each player, we assumed that the learning task wouldn't be too difficult; based on this line of reasoning, we started by deploying a basic feed-forward single layer perceptron. We used a grid search to find the best configuration of hyper-parameters (*learning_rate*, *neurons*, *batch_size* and *epochs*) and we cross validated each result (with 3 folds) but neither the precision nor the recall had reached an acceptable level.

We then moved toward a neural network with an additional dense layer, smaller than the first, but we noticed that the validation accuracy vastly underperformed the training accuracy: we incurred in overfitting, caused by the overspecialization of the neurons.
We decided to use the single layer perceptron with a slight modification: we used a dropout layer right after the input layer (before the hidden one) hoping that the mask would be able to help the neurons in learning the highly imbalanced dataset. It is important to note that we expected the network to correctly classify the 0 label (*low ranked* player) with way higher metrics than the 1 label (*high ranked* player). Our experimental results confirmed our belief.
This time the neural network behaved correctly (no overfitting of sorts, as we can see in Figure 18) and we were satisfied with the number of correctly predicted 1 labels on the validation set (with the same cross validation configuration as before).
The grid search on this model yielded us the following hyper-parameter values:

- **learning_rate** 0.01
- **neurons** 8
- **batch_size** 12
- **epochs** 40

We further tested different activation functions, momentum and dropout percentage and we settled with the following values: **activation** *relu*, **momentum** 0.001, **dropout** 0.2.

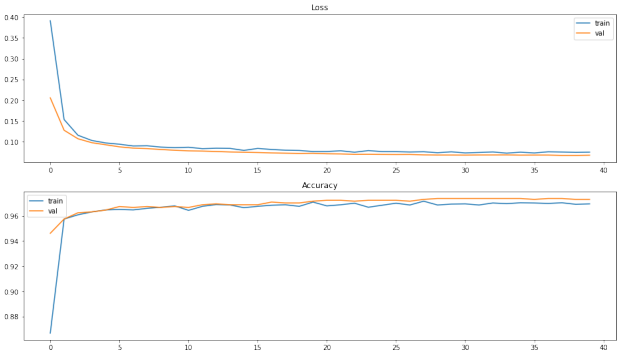

Figure 18: Loss and Accuracy on the training set wrt the validation set.

5.5 Conclusions

We tackled the classification task with three methods: a decision tree, a support vector machine and a neural network.
Contrary to the previous clustering analysis, this task belongs to the supervised learning world: records that will be used in the learning phase have a (true) label that can be used to guide the algorithm toward an understanding of the problem at hand.
Since our goal is to predict previously unseen data points, we split the dataset in two fractions: a training set (used in the learning phase) and a test set (used in the prediction). It is important to note that the test set can be used only once: we can't peek at it (neither when we standardize the data nor when we evaluate the chosen model); to select one model among many we further divided the training set in a validation set and treated the latter as a placeholder for the test data, evaluating the models there and choosing the best performing one.
The method we devised to compute the label we gave to each player created two heavily imbalanced classes: one has 8939 records and the other only 1135. Given this difference the accuracy metric isn't suitable since it does not distinguish between the numbers of correctly classified examples of different classes.
It is better to investigate the precision, recall and f1-score for each label, keeping an eye to how the training set fares with respect to the test set (to gauge the chosen model generalization capability). In the following tables, we'll report only the test evaluation.

'Random' Decision Tree			
Label	precision	recall	f1-score
low ranked	0.99	0.98	0.98
high ranked	0.84	0.90	0.87

Averaging the results we obtained with various applications of a DT with a 'random' strategy splitter we obtain near perfect metrics for the label 0 (as we can see in the Table) but lower values for the other label: we expected this behaviour since it's the least represented class.

'Best' Decision Tree			
Label	precision	recall	f1-score
low ranked	0.98	0.99	0.98
high ranked	0.89	0.84	0.87

We obtain very similar results with the DT that employs a 'best' splitter, with the only changes being a lower precision and a higher recall on the underrepresented label indicating an overall better understanding of the false positives and worse understanding of the false negatives.

SVM			
Label	precision	recall	f1-score
low ranked	0.99	0.99	0.99
high ranked	0.92	0.99	0.95

SVM with downsampling			
Label	precision	recall	f1-score
low ranked	0.95	0.99	0.97
high ranked	0.96	0.63	0.76

The results obtained by the standard SVM are even better than those we get after downsampling the dataset. We can see how reducing the number of available 'low ranked' labels reduced its precision, but at the same time the increased number of 'high ranked' increased its precision on the other label. The effect on the recall of the second label was catastrophic though: the SVM wildly overestimated the presence of 1 labels, which is reflected in the absurdly high number of false positives.

Perceptron			
Label	precision	recall	f1-score
low ranked	0.98	0.98	0.98
high ranked	0.91	0.87	0.89

The neural network results in the worst performance between all methods, notably only on the high ranked label: probably the dataset had too few records of that class to enable a correct understanding of it. Deeper networks have been explored: there were few improvements but we noticed an extremely higher risk of overfitting.
All of the three methods we investigated reported good results, both on the 'low ranked' (0) and on the 'high ranked' (1) labels, except for the SVM with downsampling.
We achieved around 0.98 f1-score for the 'low ranked' (0) label computed by all methods. A value this high can be explained by the imbalanced dataset (in favor of label 0, which constituted of almost 90% of all records) and by the easy labelling function.
The 'high ranked' (1) label prediction had a similar behaviour, with three methods (Random DT, Best DT, perceptron) hovering around the value of 0.88 and the other two methods either being either vastly superior or inferior: the SVM with downsampling yielded us a 0.76 score, while the standard SVM won the contest with the highest value of 0.95. We think that the latter method was helped by the copious amount of preprocessing that went into its construction: the deep grid search allowed the SVM to learn the differences between the low ranked and the high ranked players.
Another notable result was the constantly higher recall value (over the precision) for the 'high ranked' label for all methods except the Best DT and the SVM with downsampling (which we can consider an outlier in its own way). All in all, we would have preferred the opposite behaviour: with a lower precision we are misclassifying bad players as good, which in a real world scenario could result in inviting bad players in

extremely demanding tournaments, possibly jeopardizing their careers and/or sponsor opportunities. Our result (high recall) instead pushes us toward inviting many players, which could be good if the "overall invitation cost" (both for an abstract inviting entity that spends money and for the invitees that wage their reputation) is low, paving the way to discovering new promising players.

6 Time series analysis

In this section, we have analyzed a dataset of time series containing 100 cities with their temperature measurements from 2000 to 2009. Our task is to find groups of similar cities with respect to the temperature trends.

6.1 Removing noise

The top plot of *Figure 19* shows the average temperature for each city; we have removed the noise by applying the min-max scaler with range [-1, 1], obtaining the plot shown on the bottom.

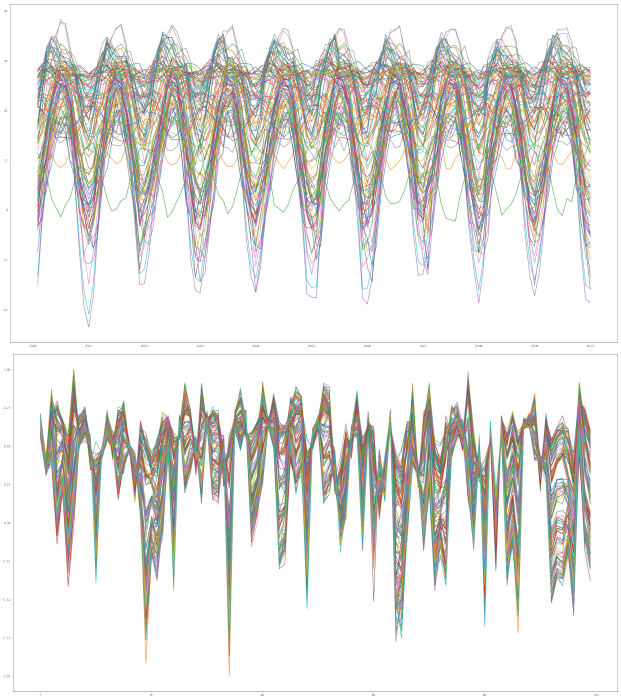

Figure 19: Time series distribution before and after removing noise operation.

6.2 Distance between time series

To analyze the dataset, we have computed the distance among some of the time series to examine the similarity of some temperature trends. In this case, we have used the **itakura** constraint.
Figure 20 shows the Euclidean distance matrix between *Baghdad* and *Wuhan*; on the background we can see the euclidean distance matrix showing the distances between the temperatures of the considered cities: each pattern represents one year worth of data. The path runs along the diagonal: we are only interested in comparing the temperature measurements taken on the same date; the rest of the matrix shows the difference between temperatures taken asynchronously, highlighting the overall changes: in this table clearer patches represent higher temperature deltas, while darker patches show smaller differences; we can appreciate how the path always threads on the darkest areas (of synchronous measurements).

The path was computed with the **dynamic time warping** method. The euclidean distance between each point of the first time series is computed with respect to all points of the second time series, saving the lowest value for each point. Then the same is done for each point of the second time series, computing the euclidean distance with respect to each point of the first one and saving the lowest value for each point. The sum of all these saved (lowest) distances represent the overall dynamic warped distance (of the path between the time series). In this case, it amounts to 1.48.

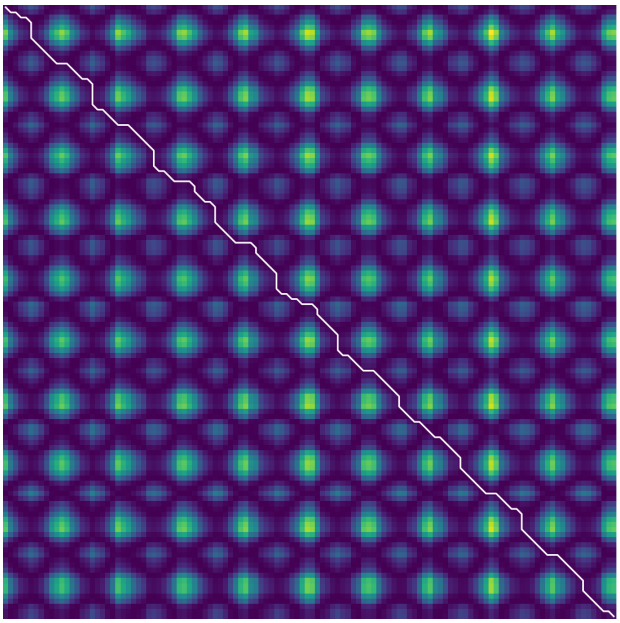

Figure 20: Path on euclidean distance matrix with the itakura constraint.

6.3 Structure-based similarity

Another approach to discover how similar two cities are is to compute some statistical features for the behaviour of the temperature for each city and then apply the euclidean distance between them. For this analysis, we computed the mean, the standard deviation, the variance, the median, various percentiles, the covariance, the skew and the kurtosis value.
In this case, the result is **0.79**.

6.4 Approximation study

To reduce the dimensionality of the data series we resorted to an approximation in a simpler space. We tested three approximations: piece-wise aggregation, symbolic aggregation and one-dimensional symbolic aggregation.

Figure 21 reveals the differences between two time series showing each approximation method. Starting from the top left we have the original time series scaled with MinMax, then going clockwise we have the three approximations. We chose two very similar cities to highlight how the approximations can help us in identifying the different behaviours which would otherwise be difficult to spot.

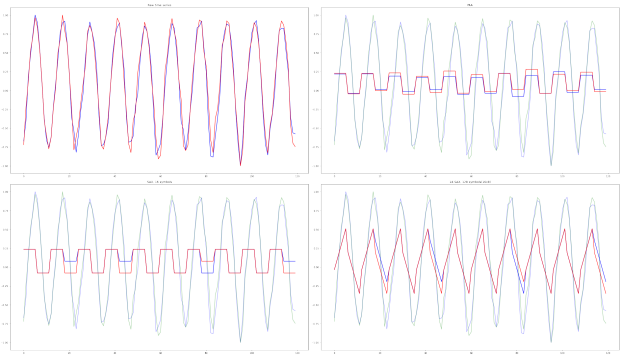

Figure 21: Approximation results between Baghdad and Wuhan.

6.5 Time series clustering

In this section, we have used clustering techniques for time series to find groups of cities with similar temperature behaviours.
We'll investigate how a shape-based clustering approach, a features-based clustering approach and a compression-based clustering approach fare against each other.

We'll use the K-means clustering algorithm in all the three approaches; we have found that the best k is **7** when using the euclidean distance (which corresponds to a 0.5 SSH value) and **9** when using the time warping distance (which corresponds to a 0.2 SSH value). In *Figure 22* we show the elbow in the euclidean distance case.

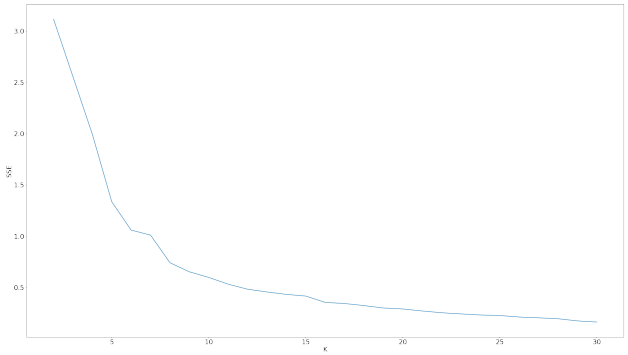

Figure 22: Graph of the elbow method.

6.5.1 Shape-based clustering

We computed the shape-based clustering with k=7 and we plotted the cluster centers to see if there are differences among the different centroids.
Figure 23 and *Figure 24* shows the different centroids; we have noticed that there is a slight overlapping on dynamic times warping metric.
Nonetheless, this metric shows better results than the Euclidean distance metric: the first one has a smaller inertia, **0.43**.

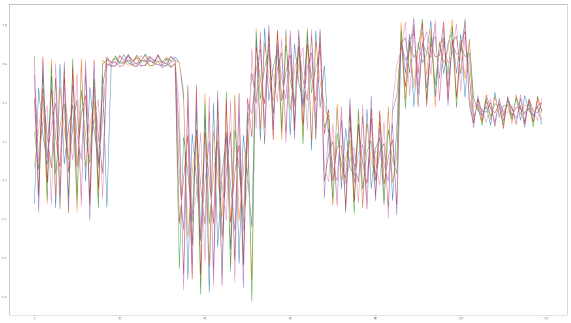

Figure 23: Shape-based clustering with Euclidean distance.

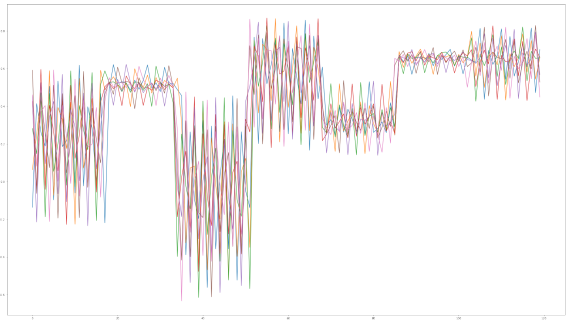

Figure 24: Shape-based clustering with Dynamic Time-Warping distance.

6.5.2 Features-based clustering

In this part, we have evaluated the clustering using statistical features (the same as those listed in **6.3**) and we have used the same number of clusters that we have found with the elbow method.
Figure 25 shows the result and we have noticed that clusters are well-separated with respect to the previous technique, but we have achieved a much bigger inertia value, **173.35**.

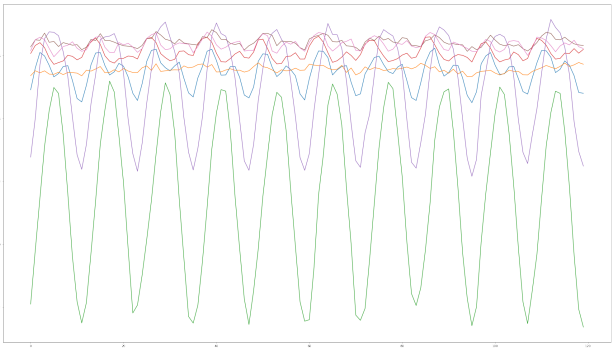

Figure 25: Clusters computed with feature-based clustering.

6.5.3 Compression-based clustering

In the last method, we have used the compression-based dissimilarity to find similar cities. Then, we have used the piece-wise aggregate approximation and K-means with 7 clusters.
Figure 26 shows that there is some overlapping among the 7 clusters.

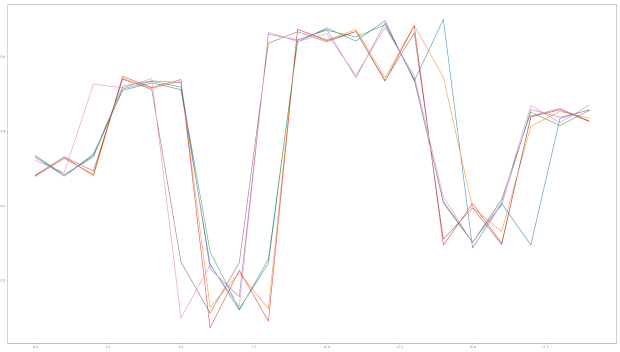

Figure 26: Clusters computed with compression-based clustering.