UNIVERSITAT POLITÈCNICA DE CATALUNYA
FACULTAT D'INFORMÀTICA DE BARCELONA

DATA MINING COURSE
DELIVERY - SECOND PROJECT

Report delivery date: 06 / 06 / 2022
Group members: Antón Calle, Nuria
Ivaylov Patchaliev, Stefan
Piotrowski, Maciej
Rodulfo Cárdenas, Alejandro
Rubio Serrano, Juan Diego
Treviño Gutiérrez, Tomàs

Contents

Introduction to the data

For this project we have searched for some datasets having at least 20 columns and 1000 rows of information. Going through Kaggle, we have found one dataset with a lot of information and good to apply our prediction models.

The chosen dataset is an Anime Dataset that already has useful information such as scores, users that have seen it, genres, number of episodes, duration, ratings… From all this information we want to try to guess what are the chances for a user to finish the series that he started. In order to do that, we will generate a new column 'completed_ratio' (which represents the ratio of users who finished the series) from the variables 'Completed', 'Watching', 'On-Hold', 'Dropped'. We divided the number of users that completed the anime by the number of users that completed the anime + are watching it + dropped it + are on-hold. We then removed those variables from the dataset so that we avoid biasing our future prediction. Then, just with all the data that has not been erased, we will try to predict the ratio of users that have finished the series. As we want it to be a classification task and not a regression one we just decided to make 4 categories representing the chances for a user to finish the serie:

- "Very Low" if the given ratio was under 0.25
- "Low" if the given percentage was between 0.25 and 0.5
- "Medium" if the given percentage was between 0.5 and 0.75
- "High" if the given percentage was over 0.75

To guess this information, first of all we have to do a series of preprocessing such as cleaning/replacing all missing data, creating the new column and erasing the used columns in order to try to guess them. Once we have done that, we will try to use all of the explained models in class to see which one has the most accuracy in our case and get some final conclusions.

Description of the original data

Dataset link : https://www.kaggle.com/datasets/hernan4444/anime-recommendation-database-2020

Original number of examples : 17558
Original number of columns : 35
Column description:

1. MAL_ID: MyAnimelist ID of the anime. (e.g. 1)
2. Name: full name of the anime. (e.g. Cowboy Bebop)
3. Score: average score of the anime given from all users in MyAnimelist database. (e.g. 8.78)
4. Genres: comma separated list of genres for this anime. (e.g. Action, Adventure, Comedy, Drama, Sci-Fi, Space)
5. English name: full name in english of the anime. (e.g. Cowboy Bebop)

6. Japanese name: full name in japanses of the anime. (e.g. カウボーイビバップ)
7. Type: TV, movie, OVA, etc. (e.g. TV)
8. Episodes': number of chapters. (e.g. 26)
9. Aired: broadcast date. (e.g. Apr 3, 1998 to Apr 24, 1999)
10. Premiered: season premiere. (e.g. Spring 1998)
11. Producers: comma separated list of produducers (e.g. Bandai Visual)
12. Licensors: comma separated list of licensors (e.g. Funimation, Bandai Entertainment)
13. Studios: comma separated list of studios (e.g. Sunrise)
14. Source: Manga, Light novel, Book, etc. (e.g Original)
15. Duration: duration of the anime per episode (e.g 24 min. per ep.)
16. Rating: age rate (e.g. R - 17+ (violence & profanity))
17. Ranked: position based in the score. (e.g 28)
18. Popularity: position based in the the number of users who have added the anime to their list. (e.g 39)
19. Members: number of community members that are in this anime's "group". (e.g. 1251960)
20. Favorites: number of users who have the anime as "favorites". (e.g. 61,971)
21. Watching: number of users who are watching the anime. (e.g. 105808)
22. Completed: number of users who have complete the anime. (e.g. 718161)
23. On-Hold: number of users who have the anime on Hold. (e.g. 71513)
24. Dropped: number of users who have dropped the anime. (e.g. 26678)
25. Plan to Watch': number of users who plan to watch the anime. (e.g. 329800)
26. Score-10': number of users who scored 10. (e.g. 229170)
27. Score-9': number of users who scored 9. (e.g. 182126)
28. Score-8': number of users who scored 8. (e.g. 131625)
29. Score-7': number of users who scored 7. (e.g. 62330)
30. Score-6': number of users who scored 6. (e.g. 20688)
31. Score-5': number of users who scored 5. (e.g. 8904)
32. Score-4': number of users who scored 4. (e.g. 3184)
33. Score-3': number of users who scored 3. (e.g. 1357)
34. Score-2': number of users who scored 2. (e.g. 741)
35. Score-1': number of users who scored 1. (e.g. 1580)

Impact of missing values : In general, the missing values weren’t a big problem in the dataset because we handle them during the preprocessing. The only missing values that bothered us was the one in the columns used to create our new “completed_ratio” feature because if one of them was missing, the value for our new feature was NaN. The number of affected rows by those is 260. Since we had more than 17000 rows, we just decided to remove them.

Description of pre-processing of data

As expected, the pre-processing took us most of the time of the project. In our opinion the dataset was especially hard to work with so we needed to do a lot of preprocessing before starting work with the dataset. Here are the different steps we did:

1. We computed the “complete_ratio” as described in the “Introduction to the data” section.
2. We removed the rows with unknown values in our “complete_ratio” feature
3. We transformed the “complete_ratio” values into 4 categories

Dropping features

4. We dropped the columns that we used to make the “complete_ratio” feature
5. We dropped the columns 'Name','English name','Japanese name' because they are directly related to 'MAL_ID'
6. We dropped the ‘MAL_ID’ column because it was just the unique ID of the anime

Dropping rows

7. We decided to remove the animes with only 1 episode because those were movies and we agreed to work only with series
8. We dropped the rows with series whose duration per episode was less than 1 minute because in our opinion it doesn’t make sense to have series with less than 1 minute per episode and we considered that as an error in the original database or as some really strange cases.

Data simplification

9. We transformed the column duration so that instead of having “24 min. per ep.” we could have “24” so that we can consider the feature as a numerical one.
10. For the “Premiered” feature, which is the start period of airing the serie, we decided to keep only the year
11. For the “Aired” feature, which is the start and end period of airing the series, we decided to keep only the end period (because we got the start already in “Premiered”) and from that to keep only the year as well.
12. We needed to replace all the “Unknown” values by np.nan so that we can apply the available nan-related python functions
13. We don’t know why but some columns that should be considered as numerical (as for example the score of a serie) were encoded as categorical so we needed to change manually the type to numeric of the following columns: Score, Score-10, Score-9, Score-8, Score-7, Score-6, Score-5 ,Score-4, Score-3, Score-2, Score-1, Episodes, Duration, Ranked

Missing values

14. Then we splitted the dataset into 2 others: one with NA values and one without. We wanted to keep 5% of rows with NA’s so that we can apply some inputting/replacement so we randomly picked 950 rows from the second dataset and 50 from the one with NA’s.
15. In the part with missing values, we decided to replace all the NA’s in the numerical columns by the median of each of those columns. For the categorical ones, we just decided to replace them by the value that occurs the most in the other rows.

Difficult features

16. One big problem that we encountered was that some features contained several values combined as 1 string value. For example, if a series belongs to 3 different

"Genres" : Action, Family and Kids, we will have just 1 "Genre" value encoded as "Action, Family, Kids". Which is a big problem if we wanted to apply OneHotEncoding afterwards because we would end up with a new feature "Genres_Action, Family, Kids" with the value 1 or 0. The problem with that is that if we have another serie with only "Action, Family" it will create a whole new feature which will be completely independent from the other one which is bad because those 2 series are at least a bit related because they belong to 2 same categories which can't be noticed with that kind of practice. Another problem is that applying directly OneHotEncoding to that yields a lot of new features (more than needed and expected). That's why we decided to apply "OneHotEncoding" for those parts by hand. Here is how we proceeded:

    a. We splitted the strings in the "Genres" feature into several ones containing different genres and created a set with the different genres
    b. We added new columns (with values 0) for each genre
    c. We iterated over all rows and over all genres contained in the "Genre" feature and set accordingly the value of that genre feature for that row to 1
    d. We deleted the old "Genres feature

17. We did the same thing for the "Licensors" and "Producers" features

One Hot Encoding & Scaling

18. We applied OneHotEncoding on the categorical features
19. We applied MinMaxScaling for the numerical features to keep the values between 0 and 1

Feature selection

20. We wanted to remove the features with 0 variance but we didn't have any. So we decided to remove the ones with a variance lower than a 0.01 threshold
21. We looked at the correlation between the variables and decided to remove the ones with a correlation higher than 0.8 or lower than -0.8.

All that preprocessing gave us a dataset with 1000 rows and 217 features. As we can notice, we end up having almost 10 times more features than the original dataset. But it's normal, it's due to the OneHotEncoding and the specific "Genres", "Producers" and "Licensors" features.

We even wanted to narrow the number of features to 50 using the SelectKBest algorithm with the chi^2 test but that left us with none of the numerical features we had at the beginning so we decided not to use it (we used it only in the SVM part)..

Evaluation criteria of data mining models

In general, we used the train_test_split method from sklearn to split the data (80% for training and 20% for testing). For SVM's we used 65% and 35% for time efficiency. Sometimes, to tune the parameters we used a GridSearchCV to test all the possible combinations of parameters that we wanted with a cv of 10 and just 1 iteration. Which is just a cross-validation on the training set with a percentage of 10. We used that number as we used it during the lab classes and agreed that we don't need more and that it works well with our dataset. We decided to go on with the F1-score accuracy metric.

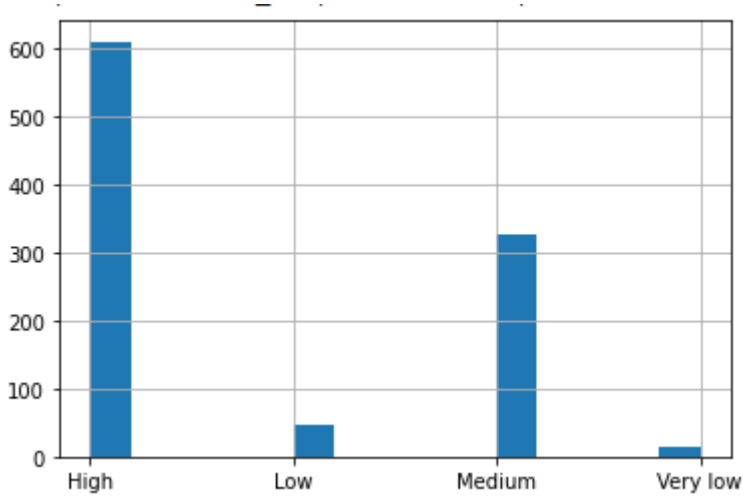
Execution of different machine learning methods

Naive Bayes

The Naïve Bayes classifier model assumes that there's independence between all the features, which is clearly not the case in our dataset, since there's several features that are more inclined to go together, such as some genres that are directly related, producers, or licensors.
Despite that, the model still manages to perform well and we get decent accuracy results. Initially we get 0.507 accuracy, but after focusing on the different classes and adjusting the threshold we conclude that focusing in either the class "Low" or "Very low" with a threshold of 0 we get the best accuracy, a 99%

K-NN

To see how our dataset behaves using a knn model, we have broken down this section into two parts. To begin with, we will apply our model to the raw dataset and see what its initial precision is and later, we will delete the features that we consider appropriate to improve the precision.
First of all, to better understand our dataset and the future results that we can expect, we have run a histogram with the "completed_ratio" target, and its distribution.

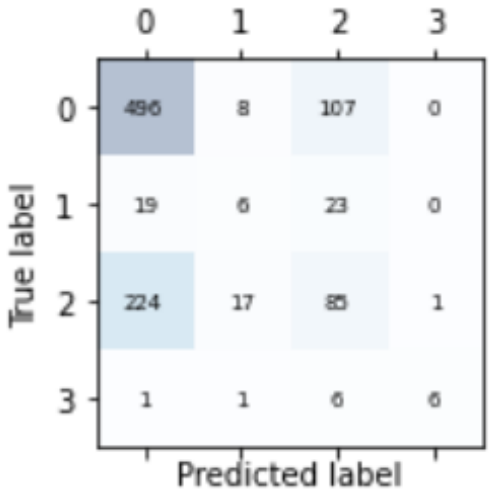


As we can see, most of our columns are distributed mainly between the "high" and "medium" categories. We will see a little later how this will affect our predictive model.

If we start by doing a simple test, we see that we only get a 0.63 accuracy. If we proceed to analyze the report of our results and we focus on the f1-score, we see that we fail a lot predicting those that are in the "low" category, since we only have 0.12. But we have to look at it too, that in "medium" and in "very low" we also failed to exceed 0.50 precision.

	precision	recall	f1-score	support
High	0.70	0.86	0.77	189
Low	0.25	0.08	0.12	12
Medium	0.39	0.24	0.30	95
Very low	0.40	0.50	0.44	4
accuracy			0.63	300
macro avg	0.44	0.42	0.41	300
weighted avg	0.58	0.63	0.59	300

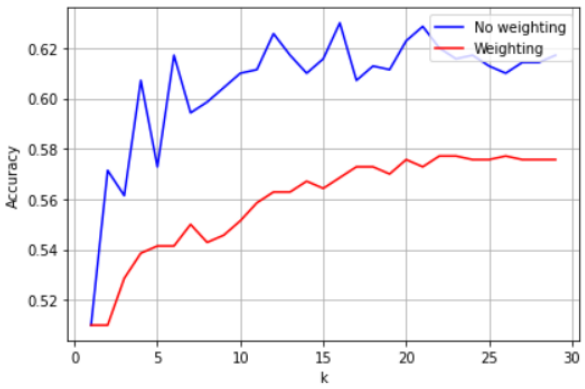
If we do a 10-cross-validation and analyze its matrix confusion, it reinforces the idea we have just explained. We have many false negatives and positives, especially in the "medium" category, since we have many values belonging to this category. This makes our precision drop down a lot, so our goal should be to try to raise our "medium" values prediction.



0.593

Now we are going to analyze which are the best parameters for our model. To do this, we are going to use gridSearch and analyze its results. It gives us that the most optimal combination is "n_neightbours" equal to 16 and "weights" equal to uniform. With these specifications we obtain a precision of 0.629.

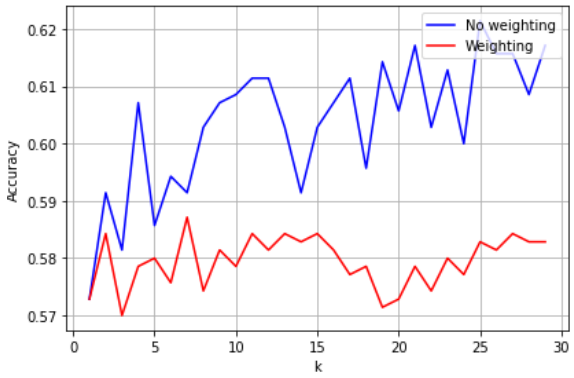
Another way to find the best parameters is to compute the 10-cross-validation and plot our results. In this way, we can also see how the variable "weight" affects our model.

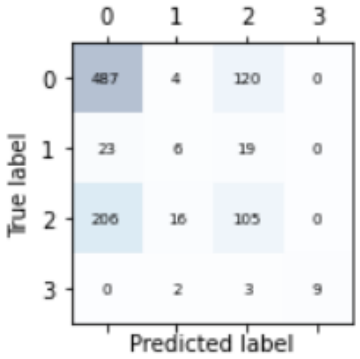


We can see that the plot gives us a result that is coherent with what we previously had and also that the highest peak is 16. In addition, we can also notice that if we add weight to our model, it is seriously affected, since it does not exceed 0.58 precision.
Now let's move on to the second part. In order to improve our precision we have calculated the correlation that our target "completed_ratio" has with the rest of the features. As we can see, most features do not provide us much value, so we opted to remove most of them. We only kept those that have a correlation greater than 20. In addition, we decided to remove the unnamed feature, which corresponds to the name of each anime, because despite having a significant correlation, it negatively affected our accuracy.

```
Unnamed: 0                                  0.241567
Score                                      -0.317140
Popularity                                  0.291362
Members                                    -0.277583
genre_Drama                                -0.032668
                                               ...
Rating_PG - Children                       -0.014617
Rating_PG-13 - Teens 13 or older            0.073650
Rating_R - 17+ (violence & profanity)      -0.087491
Rating_R+ - Mild Nudity                    -0.062524
completed_ratio                             1.000000
Name: completed_ratio, Length: 219, dtype: float64
```

If we repeat the process from before, we see how on this occasion the most optimal combination is "n_neightbours" equal to 25 and "weight" equal to uniform. With these parameters we see how our f1-score for the "medium" values increases up to 0.50 and the "high" up to 0.81, while those of low and very low decreases. As we have much less of these values, this ends up benefiting us, and we get better precision.





0.607

	precision	recall	f1-score	support
High	0.73	0.90	0.81	189
Low	0.00	0.00	0.00	12
Medium	0.60	0.43	0.50	95
Very low	0.00	0.00	0.00	4
accuracy			0.70	300
macro avg	0.33	0.33	0.33	300
weighted avg	0.65	0.70	0.67	300

So to finish with the knn model, we see that after choosing the best parameters and changing the corresponding features we obtain 0.70 final precision.

Decision Trees

A decision tree is a model which tries to classify data according to a set of features. Each branch represents a decision taken in a specific feature, and the leaves are the classes in which the data is to be classified.
In order to apply this model to our dataset, we split the data into features and target variable, which is "completed_ratio" in our case. Then, we pick the training sample (80%) and test sample (20%) in such a way that we have much more training data than testing. This part was achieved thanks to the function "train_test_split" from the sklearn library.
In what concerns the algorithm itself, we used the DecisionTreeClassifier implemented in the same sklearn library.
After an initial training and subsequent prediction of the test data, we got a 66% accuracy. That seems reasonable at first watch, however, if we predict the training data we can see that the accuracy skyrockets up to 100%. Our model is clearly overfitting.
Overfitting is a common problem in decision trees. It means that the model adjusts so precisely to the training data that it only works well in that set of data, and will underperform in any other case.
To cope with this problem, we decided to limit the maximum depth at which the tree could get to. We ran a "for" loop which tested how the model performed from maximum depth 1 to 9 with both gini and entropy selection models.

The difference between gini and entropy selection models is that the first one tries to get the leaves with the maximum purity possible (leaves with all the elements in the same class) while the later bases on the greatest entropy (-log(prob(x))).
From the results of the loop we conclude that, interestingly, gini always outperforms entropy model selection. On the other hand, the best max_depth for entropy selection model is 5 (73% test, 76.75% train, 76% total), and for the gini's, 9 (71% test, 91.75% train, 87.6% total).
It is important to note that for low values of max_depth, the model does not overfit but underfit the training set.
Finally, find here the output of the "for" loop:

Depth 1:
Entropy accuracy test: 0.62
Entropy accuracy train: 0.62125
Gini accuracy test: 0.62
Gini accuracy train: 0.62125
Entropy accuracy: 0.621
Gini accuracy: 0.621
Depth 2:
Entropy accuracy test: 0.72
Entropy accuracy train: 0.655
Gini accuracy test: 0.73
Gini accuracy train: 0.68
Entropy accuracy: 0.668
Gini accuracy: 0.69
Depth 3:
Entropy accuracy test: 0.705
Entropy accuracy train: 0.695
Gini accuracy test: 0.7
Gini accuracy train: 0.72125
Entropy accuracy: 0.697
Gini accuracy: 0.717
Depth 4:
Entropy accuracy test: 0.7
Entropy accuracy train: 0.74125
Gini accuracy test: 0.695
Gini accuracy train: 0.75
Entropy accuracy: 0.733
Gini accuracy: 0.739
Depth 5:
Entropy accuracy test: 0.73
Entropy accuracy train: 0.7675
Gini accuracy test: 0.685
Gini accuracy train: 0.775
Entropy accuracy: 0.76
Gini accuracy: 0.757
Depth 6:
Entropy accuracy test: 0.645

Entropy accuracy train: 0.765
Gini accuracy test: 0.7
Gini accuracy train: 0.80625
Entropy accuracy: 0.741
Gini accuracy: 0.785
Depth 7:
Entropy accuracy test: 0.66
Entropy accuracy train: 0.82875
Gini accuracy test: 0.68
Gini accuracy train: 0.85125
Entropy accuracy: 0.795
Gini accuracy: 0.817
Depth 8:
Entropy accuracy test: 0.69
Entropy accuracy train: 0.8275
Gini accuracy test: 0.695
Gini accuracy train: 0.87125
Entropy accuracy: 0.8
Gini accuracy: 0.836
Depth 9:
Entropy accuracy test: 0.66
Entropy accuracy train: 0.88125
Gini accuracy test: 0.71
Gini accuracy train: 0.9175
Entropy accuracy: 0.837
Gini accuracy: 0.876

Support Vector Machines

For the SVM part, we decided to perform a different splitting of the data. Instead of using 80% of the data for training we decided to use 65% which is a bit worse for training the model. But it was necessary as SVM takes a really long time to train when there are a lot of data samples. And as we wanted to try out different kernels and different parameters, reducing the size of the training set was inevitable. Also, we worked on SVM's as last so we weren't sure, if we let the 80% splitting, that we would be able to obtain all the results for all the different parameters.

To reduce even more the time, we decided to perform a dimensionality reduction using the SelectKBest method from sklearn using the chi^2 test. which selects features according to the k highest scores. We decided to drastically reduce the number of features from around 200 to 20.

We tried at first all the different kernels with some various basic parameters variation. For that, we used the GridSearchCV method from sklearn which allows us to easily test all the parameters combination passed for a model and retrieve the one reaching the best score. We noticed that all the scores are more or less the same (using the accuracy) so we chose

the RBF one because it was slightly better and it's the default one. In general, we want to prioritize the performance and the simplicity of the model. In that case the scores were approximately equal so RBF was the obvious choice.

Even though it was the best choice, with the best parameters C = 1000.0 and gamma = 1e-06, the proportion of support vectors was extremely high (75%) so the model was clearly overfitting the data. To improve that, we decided to tune even more the parameters C, gamma and tol. But it didn't decrease our number of support vectors without decreasing the score a lot.

That's why at the end we decided to go with the Linear SVM with C=100 which got a similar score but a lot lower number of support (75% -> 49%). 49% percent isn't perfect but it's the least we reached with SVM's. Playing with the tol parameter didn't help. So we just assume that our dataset is difficult to reach a lower supports number.

Meta-learning algorithms

The **meta-learning algorithms** are mainly used as a combination of different weak learning algorithms and from them we get a strong learning algorithm. We started with a voting algorithm, using our previous different methods such as KNN, decision trees and naive bayes and getting a new accuracy for a **majority voting algorithm** and a **weighted voting algorithm**.

In this case, the accuracy we get is:

Accuracy: 0.464 [Naive Bayes]
Accuracy: 0.639 [Knn (3)]
Accuracy: 0.638 [Dec. Tree]

And for the **voting algorithm** we got:

Accuracy: 0.650 [Majority Voting]
Accuracy: 0.666 [Weighted Voting]

As we can see, using this new method, we get better accuracy, but we can do even more.

Firstly, we have to do a **bagging** method, in order to reduce the variance of an estimator, as it is helpful to average estimates from independent draws from the data. After the bagging method, we get the following accuracy, for the different nests:

Accuracy: 0.637 [1]
Accuracy: 0.671 [2]
Accuracy: 0.685 [5]
Accuracy: 0.710 [10]
Accuracy: 0.729 [20]
Accuracy: 0.725 [50]
Accuracy: 0.722 [100]
Accuracy: 0.718 [200]

Our best accuracy is at the 20 nests.

Then, the next method is a **random forests** method. We are trying to combine predictor trees in statistics in which each tree depends on the values of a random vector tested independently and with the material distributed by each quest. For each nesting options we get the next accuracy:

Accuracy: 0.608 [1]
Accuracy: 0.649 [2]
Accuracy: 0.670 [5]
Accuracy: 0.679 [10]
Accuracy: 0.710 [20]
Accuracy: 0.698 [50]
Accuracy: 0.719 [100]
Accuracy: 0.719 [200]

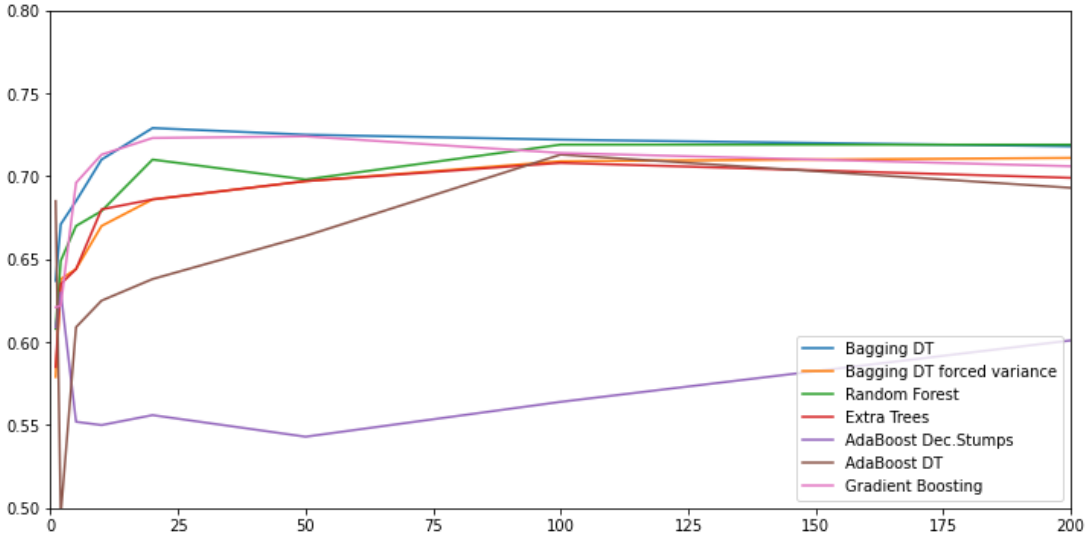
We can see that the best accuracy is during the 100 or 200 nests.

And finally we get to do the **boosting**. Boosting tries to actively improve accuracy of weak classifiers by training a sequence of specialized classified based on previous errors, creating classifiers and changing weights. For this case, we get the following accuracies and the next picture:

Accuracy: 0.621 [1]
Accuracy: 0.622 [2]
Accuracy: 0.696 [5]
Accuracy: 0.713 [10]
Accuracy: 0.723 [20]
Accuracy: 0.724 [50]
Accuracy: 0.714 [100]
Accuracy: 0.706 [200]

In this case, the best accuracy is obtained in the 50 nests.

To sum up, we got a small graph where we can see the accuracies of all the methods with all the different used nests:



From this picture we get 3 quick conclusions:

- Bagging DT is the best for little nests, is the same as the Random Forest for larger nests
- 25 nests are the best way to organize data
- Adaboost Decision Trees are by far the worse method for this case

Comparison and conclusions

Comparison of the interval of confidence of different models (95%):

Naive Bayes	K-NN	Decision Trees	SVM	Meta-Learning
[0.441, 0.579]	[0.636, 0.764]	[0.668, 0.792]	[0.590, 0.690]	[0.652, 0.778]

We can see that in our case, the decision trees reached the best accuracy score and that its confidence interval is the highest so we can say that it's our best model. As our dataset was very specific and the preprocessing part was really hard, we are very happy with the results obtained. Working with different models allowed each of us to be able to compare them and select the best fitted model for our classification task.

This project was very interesting because of multiple reasons:

- It provided a better understanding of the different subjects that we saw during the lectures
- It allowed us to deepen our working skills with Python and notebooks
- It allowed us to have a first grip on the different tools available nowadays to practice a little bit the different techniques that we saw during the lectures
- It showed us how important Data Mining is nowadays, especially with the current overload of information and why the pre-processing part is one of the most important
- It showed the difficulty of predicting, as with all this information we got as close as a 0.778 accuracy

The whole project pushed us to do a lot of web search and that, in addition to the course material, allowed us to observe all the particularity of the different techniques presented during the lectures. We can now really understand the importance of the pre-processing part just by comparing our initial dataset with the final one. The difference is huge. And not just because the dataset is trimmed but because of the data transformation as well. This project is very valuable and will surely help us for the exam preparation or some future projects related to Machine Learning.

Work repartition

Prepocessing : Maciej & Tomi
Naïve Bayes : Tomi
K-NN : Nuria
Decision Trees : Alejandro
SVM : Maciej
Meta-Learning Methods : Stefan