ELEN-0062: Introduction to Machine Learning

Project 3 - Human Activity Prediction

Maxime Goffart
180521

Olivier Joris
182113

Academic year 2021 - 2022

Contents

1 Introduction

In this report, we will present and justify the different techniques we used for the third project of the course. The goal is to provide the reasoning behind our choices and to present the results obtained on Kaggle.

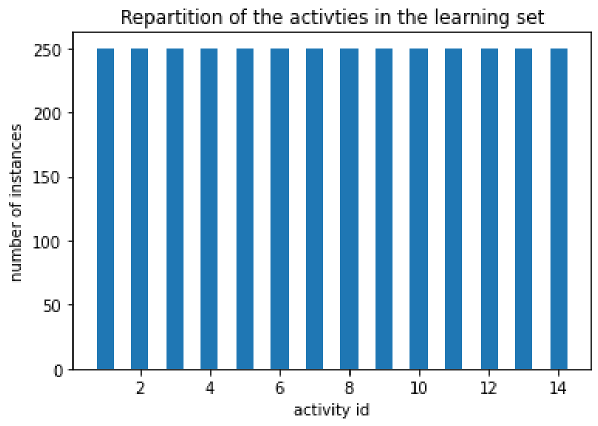
2 Data pre-processing

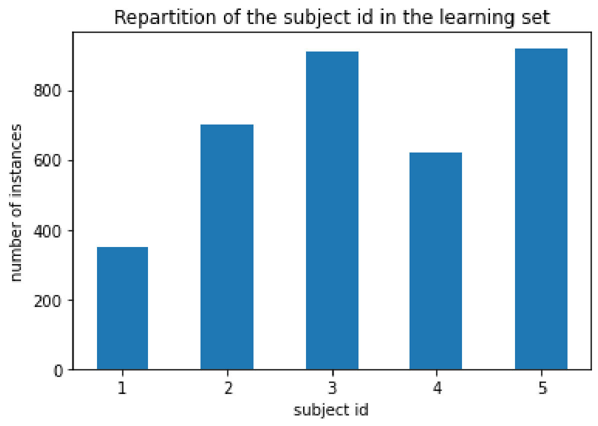
We started by analyzing the data in order to pre-process them. Indeed, pre-processing is important to have good results when doing classification. This section includes the observations we made about the dataset and different approaches we tried in order to solve the problems linked to the dataset.

2.1 Filling missing values

The learning set contained a certain number of missing values represented by the `-999999.99` value. We thus started by looking at several methods proposed by the scikit-learn library[1] to replace these missing values. We tried several of these methods and finally decided to use the `KNNImputer` which completes missing values using k-nearest neighbors : each missing value is imputed using the mean value from the nearest neighbors found in the training set.

2.2 Feature selection





We observed that the learning set seems to be balanced. We can see on the figure at the top right that the activity ID's are well distributed: they are each composed of 250 instances of the subjects in the training set. However, when looking at the subject ID's of the learning set, we observed that some subjects have been less used to perform measures. This can be observed on the figure at the top right. This can lead to overfit: all subjects have not the same behavior when performing an activity and the sensors measures are correlated to one subject.

These observations and the fact that the dataset was really huge (3500 * 31 * 512 values), motivated us to perform features selection. We decided to perform feature selection using `SelectFromModel`[2]. It is performing ranking based on one estimator and then remove the less ranked features recursively. We chose the `ExtraTreeClassifier`[3] as estimator for this method.

[1]`https://scikit-learn.org/stable/modules/impute.html`
[2]`https://scikit-learn.org/stable/modules/feature_selection.html`
[3]`https://www.geeksforgeeks.org/ml-extra-tree-classifier-for-feature-selection/`

It is similar to a random tree classifier and only differ in the manner it is constructed which lead to the creation of multiple de-correlated decision trees. This seems us to be a good model to rank the features because we have seen in the theoretical course that decision trees are nice to discriminate features but are too much related to the dataset. It is why the way the Extremely Randomized Trees are constructed reduce the variance link to the dataset and thus reduce the chances of overfitting.

2.3 Implementation

The implementation of this part can be found in the `preprocessing.ipynb` notebook or in the `preprocessing.py` python script. We have performed some tests measuring the scores and it is a bit increasing the scores of our models.

3 Studied models

3.1 K-nearest neighbors

First, we decided to use the K-nearest neighbors method because it was the one provided by the pedagogical team with the assignment. Yet, the reasons that made us trying multiple versions of this technique are the facts that it is easy to interpret and to use.

3.1.1 1-NN

Our first submission was the one provided with assignment[4] that was generated with the 1-nearest neighbor model[5].
This submission yields a score of 0.52 on Kaggle.

3.1.2 25-NN

Based on 3.1.1, we decided to increase the value of K of the K-nearest neighbors method to 25. The motivation behind the choice of this value for K was theoretical. Since the dataset is huge and can contain errors in the measured values, we though that increasing the value of K would increase the precision of the model because each prediction will not depend on a single neighbor that could have been misclassified.
This technique is implemented in the file `knn_basic_25.py`. This submission yields a score of 0.54 on Kaggle.

3.1.3 55-NN and 49-NN

We decided to keep using the K-NN method to study how well it could perform on the assignment. We decided to study the accuracy of the KNN technique by using a 10-fold cross validation technique with the negative log loss function as a scoring measure. We obtained the following graph of accuracies depending on the value of K:

[4]File `example_submission.csv`.
[5]Files `toy_script.py` or `knn_basic_1.py`.

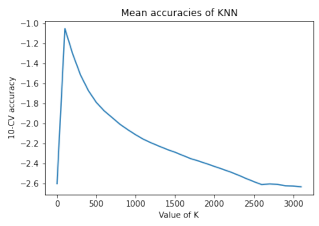


Figure 1: 10-CV accuracy of KNN depending on value of K

We could see that there is a peak. By using a divide and conquer technique, we found that the peak was obtained for values of K around 50. Thus, we decided to study the accuracy in a small range of K values around $K = 50$. We obtained the following graph:

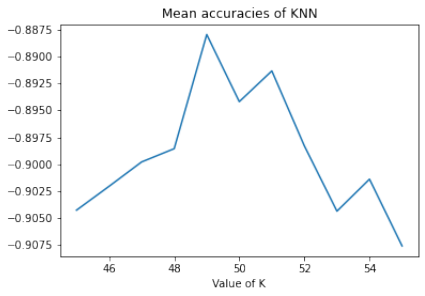


Figure 2: 10-CV accuracy of KNN depending on value of K

Based on the divide and conquer approach performed, we decided to use 55 neighbors. The choice of this value was motivated by the fact that we observed a peak around this value.
This submission yields a score of 0.53142 on Kaggle and the implementation of it is inside the file `knn_basic_55.py`.

Then, based on the graph of figure 2, we decided to use 49 neighbors. The choice of this value was motivated by the fact that it yields the highest score on the learning set by using a 10-fold cross validation strategy with the negative log loss function as a scoring measure.
This submission yields a score of 0.52571 on Kaggle which is less than the one for 55-NN. Yet, we expected a higher score based on the graph of figure 2. The implementation is available inside the file `knn_basic_49.py`.

Based on the two previous results, we came to the conclusion that using "vanilla" K-nearest neighbors was not sufficient. Thus, we decided to find new techniques explained in the following sections.

3.1.4 Multiple K-nearest neighbors models

We took a look back at the slides provided with the assignment and we focused on the features (sensors) of the datasets. We noticed that the features are not in the same units. Thus, we thought about using multiple 1-nearest neighbor models with one 1-NN model per feature. This idea was motivated by the fact that we though we would obtained a better accuracy by using one 1-NN per feature because measurements in different units would not be mixed.
We implemented this approach in the file `knn_splitted_1.py`. This yields a score of 0.52 on Kaggle which is the same as the toy script provided with the assignment.

3.1.5 Mutiple K-nearest neighbors models and samples modification

After the disappointing result obtained in 3.1.4, we decided to take a better a look at the datasets. We noticed that some samples were vectors full of -999999.99. We though that these measurements would badly influenced the models. Thus, we decided to replace each sample that was a vector full of -999999.99 by a vector full of 0.
We implemented this approach in the file `knn_splitted_1_filtered.py` and the score obtained on Kaggle was 0.53142.

3.1.6 Conclusion on K-nearest neighbors

After all the different tries perform using K-nearest neighbors models, we came to the conclusion that "vanilla" K-NN were not sufficient for this project.
The codes for the different plots are available inside the Jupyter notebooks `knn_basic.ipynb` and `knn_splitted.ipynb`.

3.2 Decision trees

Secondly, we decided to use the decision tree method because its interpretability is very easy and the effect of the parameters is easy to understand.

3.2.1 Decision tree with min_sample_split=2

First, we try a very simple decision tree with the parameter `min_sample_split` set to 2. We decided to choose this value for that parameter because we wanted to have a first result for the decision tree on which we could based our further study of the approach.
This model has been implemented in the file `dt_basic_2.py`. It has an accuracy of 0.8074285714285715 measured with a 10-fold cross validation strategy and provided a score of 0.38571 on Kaggle.

3.2.2 Decision tree with post-pruning

We decided to study the accuracy of the decision tree depending on the value of the `min_sample_split` parameter. In order to do, we decided to study the accuracy for values of the parameter ranging from 2 to 15500. We obtained the following graph:

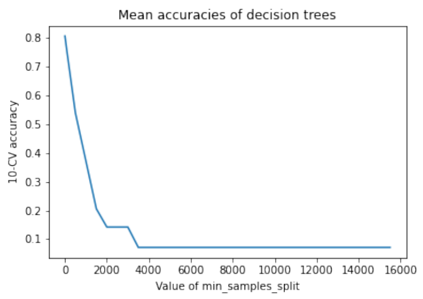


Figure 3: Accuracy of decision tree depending on value of parameter

Based on figure 3, we can see that the value of the parameter, that results in the highest accuracy, is smaller than 2000. Thus, we decided to study the accuracy of the model for values of the parameter between 2 and 2000. We obtained the following graph:

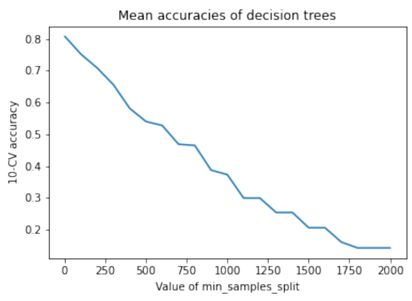
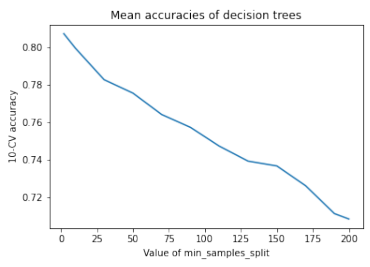
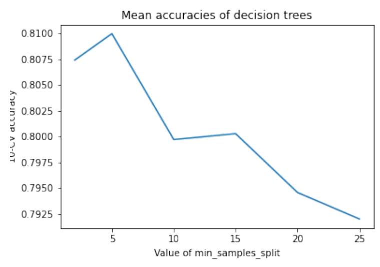
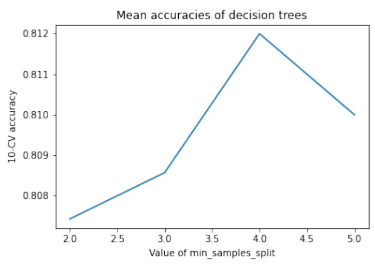


Figure 4: Accuracy of decision tree depending on value of parameter

We continue this approach and study the accuracy of the model for values of the parameter [2,200], [2,25], and [2,5]. We obtained the following graphs:

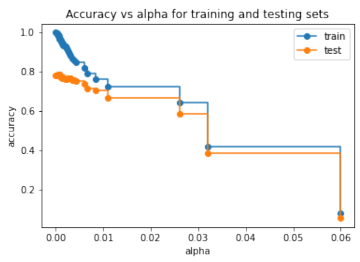


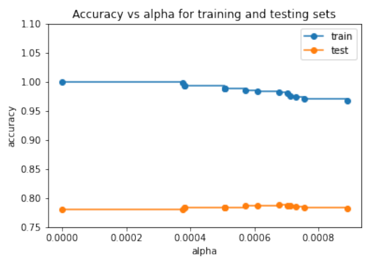




It turns out that using that using a small value for the parameter `min_sample_split` provides a higher accuracy using a 10-fold cross validation strategy. Yet, we might overfit on unseen data. Thus, we decided to study the overfitting. We based ourselves on the approach presented in the documentation of scikit-learn[6].

We split the learning sample in 2 using proportions 75% and 25% where the 75% will be our new learning set and the 25% will be our new test set. Then, we computed the path during the minimal cost-complexity pruning and we got the alphas for the sub-trees. Afterwards, we build a tree with the different values of alphas obtained in the previous step[7]. Finally, we computed the score of the LS and the TS on the different tress and plotted it. We obtained the following graphs:





Based on the obtained data, we found out that the value of alpha that provides the highest score on the test set was 0.0006772486772486773.
So, for our next model, we built a decision tree with this particular value of alpha for pruning.

[6]https://scikit-learn.org/stable/auto_examples/tree/plot_cost_complexity_pruning.html#
[7]We actually build only a fifth because there was too much possible alphas.

This particular model provided an accuracy of 0.8025714285714285 using a 10-fold cross validation approach and a score of 0.40285 on Kaggle, which is small improvement compared to the score 0.38571 obtained without pruning.
The analysis performed for this subsection is available in the Jupyter notebook `dt.ipnyb` and the implementation of the model in the file `dt_pruned.py`.

3.2.3 Decision tree with filtered data

As explained in section 2, the data provided is not ideal. Thus, we decided to try a decision tree with pre-processed data.
This model is implemented in the file `dt_filtered.py`. This particular model provided an accuracy of 0.7928571428571429 using a 10-fold cross validation approach[8] and a score of 0.43428 on Kaggle. This is an improvement compared to the 0.40285 obtained with pruning.

3.3 Decision tree pruned with filtered data

TO BE TRIED!

3.4 Random forests

Next, we decided to use random forests because they should provide better accuracies than decision trees. Yet, their interpretability is still easy.

3.4.1 Random forest with 40 trees

First, we decided to try a few forests and measuring their accuracies using a 10-fold cross validation approach. We obtained the following results:

#	**Number of trees**	**min_sample_split**	**10-CV accuracy**
1	10	200	0.7354285714285714
2	50	200	0.7702857142857142
3	50	100	0.8608571428571429
4	100	50	0.9062857142857143

Table 1: 10-CV accuracy of different random forests

Based on table 1, the first conclusion we draw was that increasing the number of trees and diminishing the value of `min_sample_split` increases the accuracy measured with a 10-fold cross validation approach.

Then, we decided to study the accuracy of the forest based on the number of trees with a fixed value for `min_sample_split` (arbitrarily set to 25). We obtained the following graph:

[8]Available in the Jupyter notebook `dt_filtered.ipynb`.

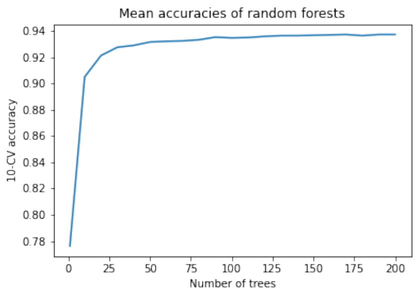


Figure 5: Accuracy of random forests depending on number of trees

Based on figure 5, we can see that, for a number of trees greater or equal to 30, we reach a plateau. Now, we will study the accuracy of random forests depending on the value of `min_sample_split` for a number of trees fixed to 40. We obtained the following graph:

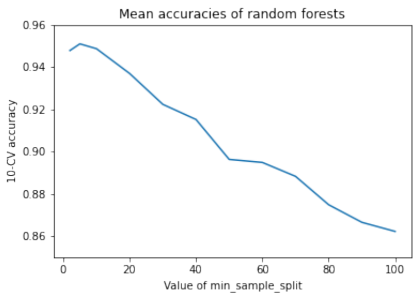


Figure 6: Accuracy of random forests depending on `min_sample_split`

As we can see on figure 6, the smaller the value of `min_sample_split`, the higher the accuracy using a 10-fold cross validation measure.

Based on a what we have observed, we decided to try a random forest composed of 40 trees with `min_sample_split` set to 25.
This model is implemented in the file `rf_40_25.py` and the analysis performed is available in the Jupyter notebook `random_forest.ipynb`.
This model provided an accuracy of 0.9288571428571428 using a 10-fold cross validation strategy and a score of 0.52 on Kaggle. This score on Kaggle is the same as the one obtained for 1-NN. Yet, this is an improvement compared to the scores obtained for the decisions tree in the part 3.2 of this report. This is what was expected because random forests use multiple trees.

3.4.2 Random forest with filtered data

As explained in section 2, the data provided is not ideal. Thus, we decided to try a random forest with pre-processed data. We kept the same values for the number of trees and `min_sample_split`.
This model is implemented in the file `rf_filtered.py`. This particular model provides an accuracy of 0.9074285714285713 using a 10-fold cross validation approach[9] and a score of 0.48285 on Kaggle. This is less than what we obtained for a random forest without filtering the data.

[9]Available in the Jupyter notebook `random_forest_filtered.ipynb`.

3.4.3 Random forest with pruning

TO BE TRIED!

4 Summary table

The following table summaries the techniques we have tried and the scores we have obtained on Kaggle.

Approach	**Kaggle score**
1-NN	0.52
25-NN	0.54
55-NN	0.53142
49-NN	0.52571
Multiple 1-NN	0.52
Multiple 1-NN with filtered data	0.53142
Decision tree - min_sample_split=2	0.38571
Decision tree with pruning	0.40285
Decision tree with filtered data	0.43428
Random forest - 40 trees - min_sample_split=25	0.52
Random forest - filtered data	0.48285

Table 2: Approaches with respective Kaggle scores