Machine Learning Challenge
Semester: 2020/21-2

Gábor Vitrai - ABIOWE

[1] University of Eötvös Loránd, Budapest, Hungary http://www.inf.elte.hu
[2] Faculty of Informatics

Abstract. This document was created for the Machine Learning Subject as a final - end of year - assignment/challenge. The aim of the project is to analyse the data, and select the most suitable Machine Learning model and predict the labels for the test data set.

Keywords: Data Science · Classification · Prediction · Random Forest

1 The Data Set

Data Analysis Upon first receiving the data set we can see, that it mostly contains random data. Unfortunately, we can not conclude to an included problem, like in the case of a Heart Disease prediction exercise. After reviewing the data, with the help of some basic built in functions we can easily get some primary information of the records. We can see that we have 400 instances in the train and 100 samples in the test set. The tables we got, do not include a header, so in order to use these I generated a header with different markings for numerical and categorical data. The result was the following markings: C1 C2 C3 C4 ... X1 X2 X3 X4, where the attributes starting with 'C' are categorical data, while the attributes starting with "X" are the numerical data.

Table 1. Count of labels in the Training set

Label	Count	Percentage
-1 - Negative Label	**237**	59,25%
1 - Positive Label	**163**	40,75%

Numerical Data: The data set can be divided into two category in the meaning of attributes. The basic numerical columns are easier to deal with. These are simple values, so it is not needed to encode the data, we only have to think about scaling the data. Luckily, the data set was provided with normalized data so there was no need to use a scaler this time.

Categorical data: We can see that the remaining categorical columns are encoded. This is marked with an "Cx" format, where "x" are integers. Some sort of encoding of the categorical variables will be required.

2 The Models

In this section, I will write about the process of creating, testing, evaluating and fine tuning different models. First we divide the data into two groups. The **target** (which is a vector) is containing values 1 or -1. This shows us how we classify the test instances to the "Positive" or "Negative" category. We will use the **training** data to train our model, so it can predict these result by itself. The **features** matrix is the data, consisting of every attribute except the target labels.

2.1 Training with the Test Split

Imputing the missing data: The dataset is full of missing values marked with NA. This makes predictions impossible as it is not suitable for models to learn from. To get rid of these, we have to use some sort of Imputer(s) to replace the unknown data. The Imputers will replace the missing values with - for example - mean, median or most frequent values, so the statistical values will not change dramatically. Of course this requires some test and fine tuning.

First, I used a SimpleImputer from sklearn for both the numerical and categorical data, with 'mean' strategy for numerical and 'constant' for the categorical. Then I decided to change the 'constant' to most_frequent, as this gave a better result.
As the figure shows below, i was not happy with the imputed numerical data, so I changed it for an Iterative Imputer. This estimates each feature from all the others. I decided to go with 'median' strategy upon imputing. These results were much better, as the 3rd part of the figure shows below.

Fig. 1. Original - SimpleImputer - IterativeImputer

X10	X11
0.820	0.352
-1.823	-1.933
-1.284	-0.762
-0.546	-0.505
-1.014	0.409
...	...
NaN	NaN
NaN	NaN
NaN	NaN
NaN	NaN
NaN	NaN

X10	X11
0.820000	0.352000
-1.823000	-1.933000
-1.284000	-0.762000
-0.546000	-0.505000
-1.014000	0.409000
...	...
-0.025547	-0.031954
-0.025547	-0.031954
-0.025547	-0.031954
-0.025547	-0.031954
-0.025547	-0.031954

X10	X11
0.820000	0.352000
-1.823000	-1.933000
-1.284000	-0.762000
-0.546000	-0.505000
-1.014000	0.409000
...	...
0.012301	0.058561
0.106467	0.248733
0.271979	0.106961
0.358153	0.200978
-0.055968	0.019857

Encoding: We have to encode the categorical data. For this, we use OneHotEncoding. It is a natural encoding for ordinal variables. By default, it will assign integers to labels in the order that is observed in the data, which is perfect for us. To do this, I had 2 options. The first one is to use Pandas built in function get_dummies(), or use the OneHotEncoder() from the sklearn library. Last semester I used the one from the sklearn library, so to try this one as well, I went with the built in function. To make sure, that from the train and test set, the method will generate the same dummy columns, I concatenated them, called the method, and then divided them into their original variables.

Fig. 2. Original - OneHotEncoded data

C4	C5
V1	V1
V2	V2
V1	V1
V1	V1
V2	V2
...	...
V1	V3
V2	V2
V1	V2
V2	V2
V1	V1

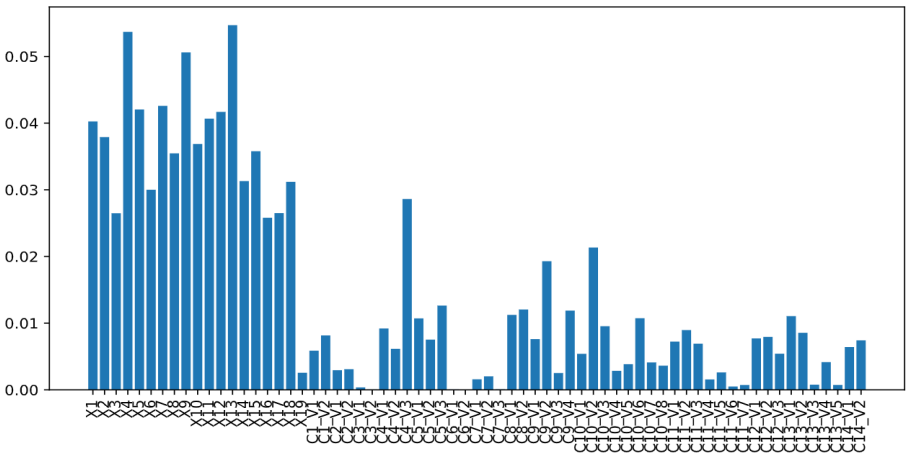
C4_V1	C4_V2	C4_V3	C5_V1	C5_V2	C5_V3
1	0	0	1	0	0
0	1	0	0	1	0
1	0	0	1	0	0
1	0	0	1	0	0
0	1	0	0	1	0
...	...	...	...	...	...
1	0	0	0	0	1
0	1	0	0	1	0
1	0	0	0	1	0
0	1	0	0	1	0
1	0	0	1	0	0

We can see how the C1 attribute got divided into 2: C1_V1, C1_V2. This happened with every categorical data. (See the figure as example) At this point, I noticed that with OneHotEncoding, we have a lot of columns. To reduce the number of columns and to remove not necessary data, I used some feature selection.

Additional Thoughts about the data set: The plan is the following: Use the train.csv file, split it into train and test sets, so we can select, test and fine tune our model which we will make our predictions with. For the final prediction, I will use the same mode, we fine tuned, and train it on the full train.csv dataset to predict the content of the test.csv file.

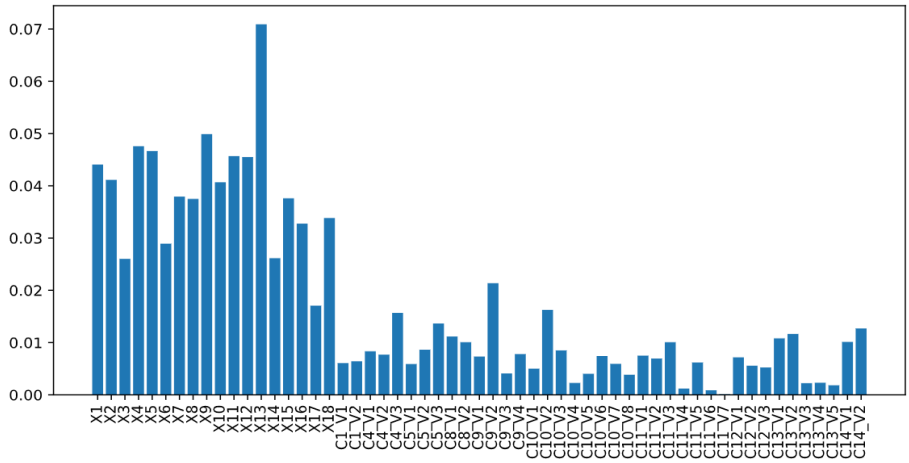
Feature Selection: We have many features, this can result in an uncertain model. After decoding the categorical data, I used a base model of a RandomForestClassifier, to get and plot the original feature importances.

Fig. 3. Original Feature Importances



Based on the plot, we can see one numerical attribute with less then 0.0025 importance, which I filtered out from the data set. In the case of categorical attributes, based on the scores of the plot, I removed the C2, C3, C6, C7 columns. This removal increased the test score by an average of 1,5 - 2 percent. After the removal, the feature importances looked like the following.

Fig. 4. New Feature Importances



At this point, we have the imported, examined, splitted, scaled, imputed, encoded, selected features. The pre-processing is done. Next step is to select, train and optimize the most suitable model. We can see that the categorical

data columns seems to have less importance, but if we would merge the columns in the same category, we would get columns with similar heights/importance, as in the case of numerical data.

Training: With this **transformed** data we can start the training. The following code will split the matrices into random train and test subsets. With the "test-size" parameter we can set what proportion of the data to include in the test split. I used 20% in most cases during the testing, as the original files has a 1 to 4 ratio, but I did not want to go higher then 20%.

```
target = df["Y"]
X_train, X_test, y_train, y_test =
train_test_split(features, target, test_size=0.2)
```

2.2 Selecting the best Classifiers

I started with 5 different models. **LR, SVM, KNN, BAG, RF**. I used Repeated Stratified KFold to cross validate and determine the overall basic scores of these models. Logistic Regression, K Nearest Neighbor Classifier and the Support Vector Machines could not overcome the 72%-75% accuracy, while the Bagging Classifier Started to give better scores upon cross validation as it was an ensemble method. This is due to the random nature of the data set. The next model which could even reach the 80%-81% score was the Random Forest Classifier.

2.3 Random Forest Classifier

As we have randomized data, based on the previous inspections, I decided to go with a Random Forest Classifier base model first. A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the data set and uses averaging to improve the accuracy. This model, achieved around 80% accuracy on the test set, which is quite good, yet sometimes, depending on the random seed of the splitting, this could differ. Naturally, I kept an eye on other metrics like precision, recall, f1-score, the confusion matrix, but since the assignment serves as a challenge based on the accuracy score, I mainly checked the Test score and the confusion matrix.
The confusion matrix is a handy visualization of the numbers our model's scores and metrics are calculated from. The base model looked the following:

```
[[37  7]    [[TN FN]
 [ 7 29]]    [FP TP]]
```

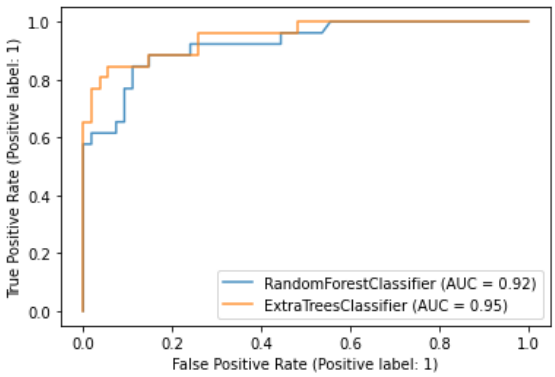
This shows, that out of 44 positive instances, our model successfully predicted 37, while out of 36 negatives it predicted 29 correctly. This resulted in a 82,5% test score. The count of true negatives are in the upper left corner, false negatives are in the upper right corner, true positives are in the lower right corner, and the false positives are in the lower left corner.

2.4 Extra Trees Classifier

This classifier is very similar as the Random Forest as both are ensemble methods, which are using many decision trees, where the final decision is obtained taking into account the prediction of every tree. The differences: Random forest uses bootstrap replicas, that is to say, it sub-samples the input data with replacement, whereas Extra Trees use the whole original sample. However later in experimenting I realized (More about this later), that the recommended number of subset in the case of our data set is 2 or rarely 3, so I decided to involve the Extra Trees Classifier which selects the subsets randomly, giving a more generic prediction. Random Forest chooses the optimum split while Extra Trees chooses it randomly. However, once the split points are selected, the two algorithms choose the best one between all the subset of features. Therefore, Extra Trees adds randomization but still has optimization. This classifier tended to provide a little bit better results, with better prediction in the case of an average 1-3 instances.

```
[[38  6]
 [ 5 31]]
```

Fig. 5. True Positive - False Positive (Two Base models)



2.5 Optimizing the classifiers

Both of the classifiers have very similar attributes, so i could use the same optimization method for both classifiers.

Randomized Search CV I decided to try to tune the hyper parameters of the Random Forest and Extra Trees Classifiers. For this, I had to find the the proper attributes for these 6 hyper parameters: n_estimators (number of estimators), max_features (either 'auto' or 'sqrt'), max_depth (maximum depth of the trees), min_samples_split (minimum number of samples of the split), min_samples_leaf, bootstrap (Boolean, if we want to have splitting of the set) I did some research and I found a random grid with these parameters. This would require 4320 settings to be tested, which is a lot. Because of the amount of tests needed to be done, i used a Randomized Search CV as taking all of these would take a lot of time.

Grid Search CV With the output of the Randomized Search CV, I could reduce the required numbers of tests needed. This reduced number of combinations so I could run these with a Grid Search CV, which is an exhaustive search over specified parameter values for an estimator. Here I specified a dictionary with the requested parameters, to which we can train my model. The execution will result in the recommended estimator. For our model, the recommended parameters were the following which resulted in an average of 2% improvement.

```
{'bootstrap': False,
 'max_depth': 60,
 'max_features': 'auto',
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'n_estimators': 1500}
```

Voting Classifier I was still not happy with the results, so I decided to use a Voting Classifier which is a Voting/Majority Rule classifier for estimators. I Experimented with it, and I decided to use 'hard' voting with weights to the different estimators. I added a Random Forest Classifier, a Bagging Classifier and an Extra Trees Classifier plus the Fine tuned Extra Trees Classifier. The Last one had a bigger weight as it was tuned, but I wanted to have at least one base model classifier. I also decided to give a weight of 2 to the RF classifier, as in some case, it has a better score in some splits. The Bagging Classifier also improved the overall score, when I gave it the some weights (1). So now that the final is done, I trained this Classifier with the whole train.csv and predicted the test.csv instances.

Cross validation is a model validation technique to test how the results of a statistical analysis will generalize to an independent data set.
I used this technique to validate the score of the Classifier. The Cross Validated RF Classifier had an original accuracy of **0.802** while the voting classifier had an overage score of **0.837**. Without Cross Validation, in case of some splits, the model could reach 92% accuracy.

References

1. SKLearn - Voting Classifier, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html. Last accessed 27 May 2021
2. Hyperparameter Tuning for Random Forest Classifier, https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74. Last accessed 20 May 2021
3. Bagging, https://towardsdatascience.com/using-bagging-and-boosting-to-improve-classification-tree-accuracy-6d3bb6c95e5b. Last accessed 27 May 2021
4. SKLearn - ROC Curve, https://scikit-learn.org/stable/visualizations.html. Last accessed 27 May 2021