Zsófia Sára Budai
U6A7FV
Kaggle name: Budai Zsófia

Data Mining and Machine Learning - Assignment

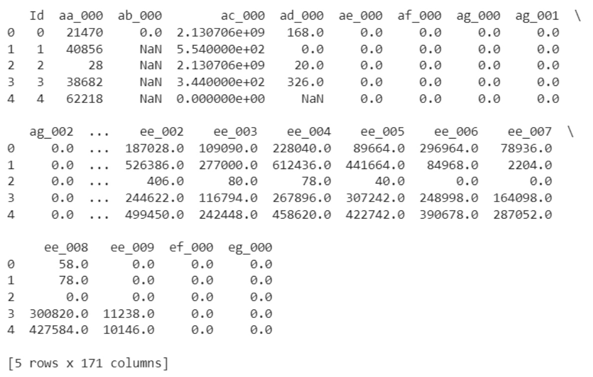
final version

1. Introduction

The Air Pressure System (APS) is an essential part of a heavy duty vehicle which generates pressurized air that is utilized in various functions in a truck. The dataset consists of data collected from heavy Scania trucks in everyday usage based on what the classification task is to decide whether a given failure is related to its APS or not. The dataset includes information on the type and performance of the APS, as well as related maintenance and repair data. It includes features such as the type of APS, the age of the system, the number of repair sessions, the number of miles driven, and the time since the last repair. This is a two class, binary classification problem where the positive class in the dataset corresponds to component failures for a specific component of the APS system and the negative class corresponds to failures for components not related to the APS. A false negative prediction is when a system incorrectly predicts that there is no problem when in reality there is a problem. This has the potential to cause more harm because if the problem is not caught in time, it can cause more damage and possibly even put people's lives at risk therefore F3 score is used to evaluate the results.

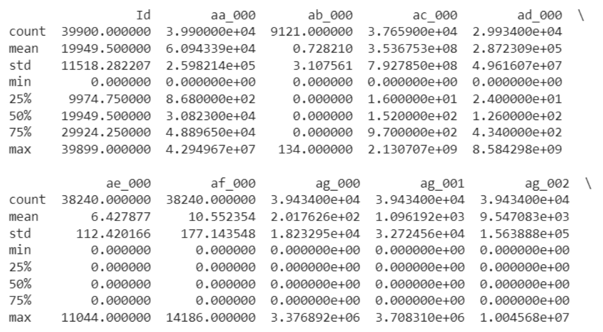
2. The Dataset

The datasets were loaded by using the pandas library in python. Pandas' read_csv() method allowed to specify the values that should be considered as missing data (i.e. NA values). In order to get to know the data better, I used dataframe.head() and dataframe.describe() to get the basic idea about the attributes. The number of features were counted which indicated 171 columns representing different attributes including the Id.
x_train: 39900 rows and 171 columns
y_train: 39900 rows and 2 columns
x_test: 174304 rows and again 171 columns



```
   Id  aa_000  ab_000        ac_000  ad_000  ae_000  af_000  ag_000  ag_001  \
0   0   21470     0.0  2.130706e+09   168.0     0.0     0.0     0.0     0.0
1   1   40856     NaN  5.540000e+02     0.0     0.0     0.0     0.0     0.0
2   2      28     NaN  2.130706e+09    20.0     0.0     0.0     0.0     0.0
3   3   38682     NaN  3.440000e+02   326.0     0.0     0.0     0.0     0.0
4   4   62218     NaN  0.000000e+00     NaN     0.0     0.0     0.0     0.0

   ag_002  ...    ee_002    ee_003    ee_004    ee_005    ee_006    ee_007  \
0     0.0  ...  187028.0  109090.0  228040.0   89664.0  296964.0   78936.0
1     0.0  ...  526386.0  277000.0  612436.0  441664.0   84968.0    2204.0
2     0.0  ...     406.0      80.0      78.0      40.0       0.0       0.0
3     0.0  ...  244622.0  116794.0  267896.0  307242.0  248998.0  164098.0
4     0.0  ...  499450.0  242448.0  458620.0  422742.0  390678.0  287052.0

     ee_008   ee_009  ef_000  eg_000
0      58.0      0.0     0.0     0.0
1      78.0      0.0     0.0     0.0
2       0.0      0.0     0.0     0.0
3  300820.0  11238.0     0.0     0.0
4  427584.0  10146.0     0.0     0.0

[5 rows x 171 columns]
```

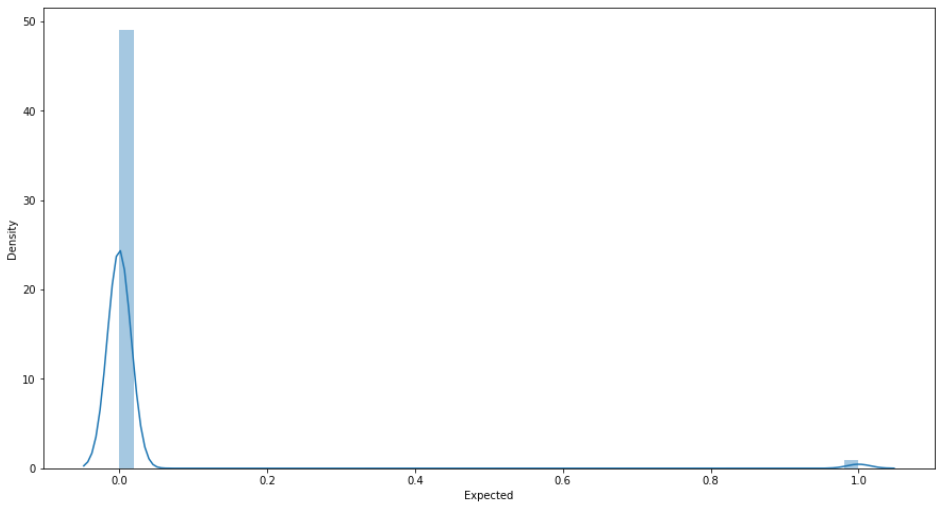
dataframe.head()



```
                 Id        aa_000       ab_000        ac_000        ad_000  \
count  39900.000000  3.990000e+04  9121.000000  3.765900e+04  2.993400e+04
mean   19949.500000  6.094339e+04     0.728210  3.536753e+08  2.872309e+05
std    11518.282207  2.598214e+05     3.107561  7.927850e+08  4.961607e+07
min        0.000000  0.000000e+00     0.000000  0.000000e+00  0.000000e+00
25%     9974.750000  8.680000e+02     0.000000  1.600000e+01  2.400000e+01
50%    19949.500000  3.082300e+04     0.000000  1.520000e+02  1.260000e+02
75%    29924.250000  4.889650e+04     0.000000  9.700000e+02  4.340000e+02
max    39899.000000  4.294967e+07   134.000000  2.130707e+09  8.584298e+09

             ae_000        af_000        ag_000        ag_001        ag_002  \
count  38240.000000  38240.000000  3.943400e+04  3.943400e+04  3.943400e+04
mean       6.427877     10.552354  2.017626e+02  1.096192e+03  9.547083e+03
std      112.420166    177.143548  1.823295e+04  3.272456e+04  1.563888e+05
min        0.000000      0.000000  0.000000e+00  0.000000e+00  0.000000e+00
25%        0.000000      0.000000  0.000000e+00  0.000000e+00  0.000000e+00
50%        0.000000      0.000000  0.000000e+00  0.000000e+00  0.000000e+00
75%        0.000000      0.000000  0.000000e+00  0.000000e+00  0.000000e+00
max    11044.000000  14186.000000  3.376892e+06  3.708310e+06  1.004568e+07
```

dataframe.describe()

By checking the distribution of our dataset, it is visible that the data is highly imbalanced as the negative class (zero value) is over represented with 39178 to 722.



Distribution of our dataset

3. **Preprocessing**
   a. **Missing data**
   The percentage of missing data for each feature in the training dataset was calculated. The features with the highest percentage of missing data in descending order are listed below:

	column_name	percent_missing
br_000	br_000	81.545894
bq_000	bq_000	80.579710
bp_000	bp_000	79.033816
ab_000	ab_000	77.198068
cr_000	cr_000	77.198068
bo_000	bo_000	76.521739
bn_000	bn_000	72.657005
bm_000	bm_000	65.700483
bl_000	bl_000	45.700483
bk_000	bk_000	39.130435

   I've tried to drop features in the training dataset that had more than 50% of their values missing by using the dataframe.drop() command. This allowed us to reduce the size of the dataset and focus on the features that had more complete data but gave better accuracy to do PCA instead of dropping. Therefor I've used the following technique for NAN values:

   b. **Simple Imputer**
   In the preprocess I have implemented the SimpleImputer from sklearn.impute that is used to impute (replace) missing values in a dataset. It does this by computing a set of statistics (mean, median, mode, etc.) from the non-missing values in the data, and then using these statistics to replace the missing values. I've tried different strategies. For most frequent the accuracy was around 74% for median 53% and for mean 73% The chosen strategy was 'most_frequent'. Important part is to do this

method also for the test data set to be able to implement PCA on that as well.

4. **Compensation for unbalanced data**
   This dataset overall was rather small and biased towards the negative class as seen in the previous subsection, it has approximately 98% data points belonging to class 0 and 2% data points belonging to class 1 which creates an unbalanced dataset. SMOTE (Synthetic Minority Oversampling Technique) was used to synthetically create additional data points from the minority class (class 1). It is synthesizing new minority class examples rather than just replicating the existing minority class examples. It works by selecting a minority class sample and finding its nearest neighbors. It then creates new samples by linearly combining these neighbors, with a random coefficient between 0 and 1.

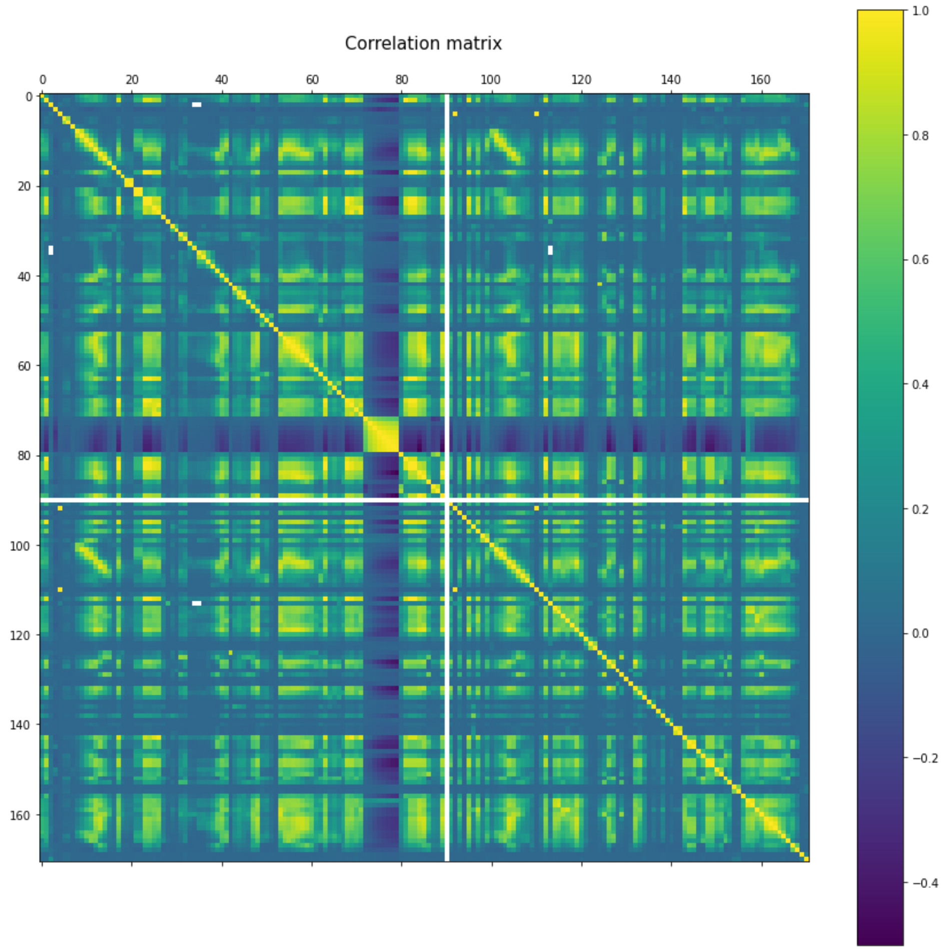
   New shape : (52494, 50) (52494,)

k_neighbors	private score	public score
5	0,75135	0,74012
6	0,75836	0,0,78281
7	0,76636	0,77524
8	**0,76841**	**0,78078**
9	0,75765	0,76967
10	0,75599	0,77058

5. **Feature extraction and dimensionality adjustment**

   The dataset initially contained 170 different features which were probably correlated and hence not required. I used Principal Component Analysis (PCA) which is a useful technique for reducing the dimensionality of a dataset. PCA is an unsupervised technique, so it doesn't take into account any labels or target variables in the data. Principal components are ranked by the amount of variation they capture in the data, and the most important ones are selected for the new feature set. By using only the most important principal components, I was able to reduce the dimensionality of the data while still retaining most of the information.

   A heatmap showing the correlation between the features is shown in the *Correlation matrix* figure. The correlation between the different attributes was also observed with a correlation matrix. The plot clearly shows that, in 171 attributes, the attributes numbered from 70 to 80 are negatively correlated with other attributes.



The PCA components were found using the decomposition.PCA(*number_of_components*) command in scikit-learn library. To begin, the value of *number_of_components* was set as 0.80 i.e. the number of PCA components that were selected should contain at least 80% of the variance of the dataset. The number of components that contained 80% of the variance.

PCA	private score	public score
90	0,76359	0,77601
80	0,76195	0,77692
70	0,75141	0,77101
60	0,75404	0,75697
50	**0,76562**	**0,76742**
40	0,75038	0,76659

6. Evaluation of the models

a. Splitting the dataset

Only the input values of the test dataset were provided, so in order to test how does the model perform in the competition it has to be submitted.To overcome this limitation and be able to test multiple models and test locally on the provided data the training data has been splitted using the train_test_split function from the sklearn.model_selection library. The optimal way of splitting the dataset is by making 1/3 of the training set a test set which will not be used in training the model, and only the remaining 2/3 of the original training set will be used for training. Also important that each class should be represented with an equal proportion in both datasets.

b. Accuracy score differences

To be able to measure accuracy locally without Kaggle I've used F3 score accuracy calculation. F-score is calculated from the precision and recall of the test. The F3 score is a metric that combines precision and recall, and it is often used in situations where there is an imbalance in the class distribution of the data. It is defined as the harmonic mean of precision and recall, with a higher value indicating a better performance. The F3 score is computed as follows:

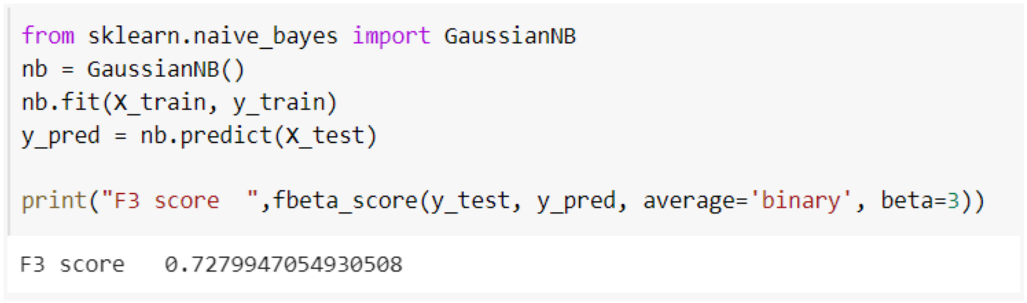
F3 = (3 * precision * recall) / (2 * precision + recall)

For this I've used fbeta_score from sklearn.metrics with average='binary', beta=3. Based on my observation this gave a really good estimation compared to Kaggles private score (public score was always a bit higher than the private).

7. Basic Algorithms

a. Gaussian Naïve Bayes:

This classifier is a probabilistic model that makes use of Bayes' theorem to predict the probability of a given data point belonging to a particular class. It assumes that the features of the data are independent of one another and are normally distributed, which simplifies the calculations required to make predictions. It is also fast and efficient, making it a popular choice for many applications. However, it can be less accurate than other classifiers in certain situations, particularly when the assumption of independence between features does not hold.



```
from sklearn.naive_bayes import GaussianNB
nb = GaussianNB()
nb.fit(X_train, y_train)
y_pred = nb.predict(X_test)

print("F3 score  ",fbeta_score(y_test, y_pred, average='binary', beta=3))
```

```
F3 score   0.7279947054930508
```

Private score: 0.71616

Public score: 0.76106

b. Regression

This is a supervised learning commonly used for numeric prediction, which is similar to classification.

```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression()
lr.fit(X_train, y_train)
y_pred = lr.predict(X_test)

print("F3 score  ",fbeta_score(y_test, y_pred, average='binary', beta=3))
```

```
F3 score   0.7006005147269089
```

Private score: 0.70351
Public score: 0.70837

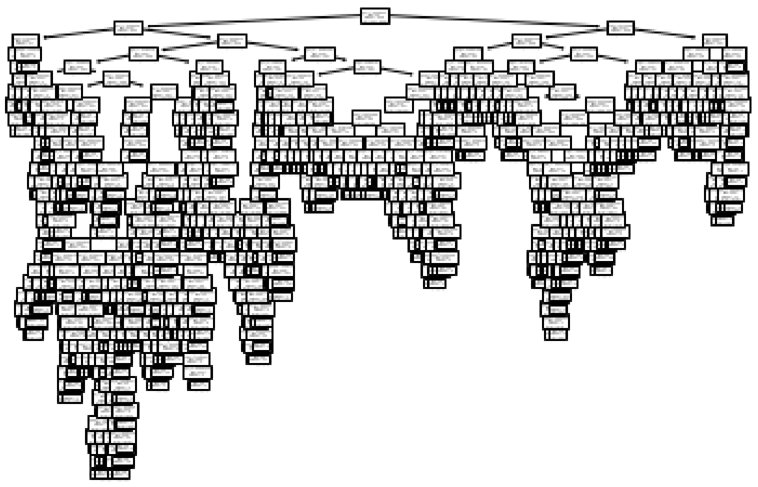
Logistic regression gave approximately the same accuracy as Naive Bayes, but it did not converge. With RandomizedSearchCV I was able to improve the accuracy only with 1%.

c. Decision Trees

This is a flowchart-like tree structure that an algorithm uses to make decisions based on a set of conditions. It is a supervised learning algorithm that can be used for both classification and regression tasks. In scikit-learn, the decision trees are implemented using the CART (Classification and Regression Tree) algorithm, which is a pre-pruning method. This means that the tree is stopped from growing beyond a certain size based on a predetermined stopping criterion, such as a minimum number of samples in a leaf node or a maximum depth of the tree. This can help to prevent overfitting and improve the generalization performance of the model.

```
from sklearn import tree
model = tree.DecisionTreeClassifier()
model.fit(X_train, y_train)

y_pred = model.predict(X_test)

print("F3 score  ",fbeta_score(y_test, y_pred, average='binary', beta=3))
```

```
F3 score   0.6652283351312478
```



Private score: 0.66326
Public score: 0.74675

The Decision Tree Regressor has also been tested, and it actually gave a slightly higher accuracy score than the Decision Tree Classifier.
Private score: 0.68616
Public score: 0.68196

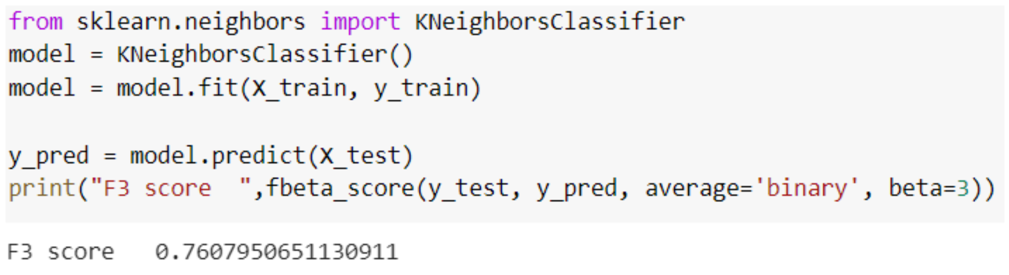
I've tried the criterion to friedman_mse in the Decision Tree Regressor and the result is visible in the following table:

criterion	**Private score**	**Public score**
squared_error (default)	0.64095	0.70006
friedman_mse	0.66966	0.61211

Changing the criterion to absolute_error and poisson made a significantly longer calculation time.

d. k-nearest neighbors (kNN)

This is a non-parametric, lazy learning algorithm that can be used for classification and regression tasks. It works by storing all of the available data, and then classifying new data points based on their similarity to the stored data. To classify a new data point, the kNN algorithm selects the k data points in the training set that are most similar to the new data point, based on some distance measure



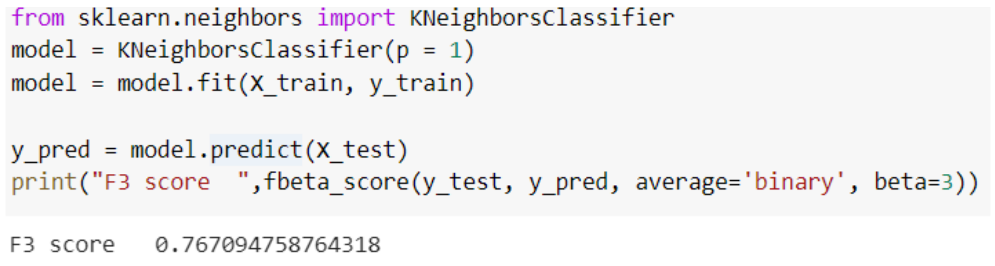
```
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier()
model = model.fit(X_train, y_train)

y_pred = model.predict(X_test)
print("F3 score  ",fbeta_score(y_test, y_pred, average='binary', beta=3))
```

```
F3 score   0.7607950651130911
```

Private score: 0.73197
Public score: 0.7625

While tuning the parameters of the classifier, setting the weight function used in prediction from the default „uniform" to „distance" increased the accuracy.
Private score: 0.73684
Public score: 0.74285

It also seemed to work slightly better when I changed the power parameter (p) from the default 2 to 1, which means that the classifier used Manhattan distance (l1) instead of Euclidean (l2). Although it made the running time longer.



```
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(p = 1)
model = model.fit(X_train, y_train)

y_pred = model.predict(X_test)
print("F3 score  ",fbeta_score(y_test, y_pred, average='binary', beta=3))
```

```
F3 score   0.767094758764318
```

Private score: 0.75
Public score: 0.73139

Important parameter in this algorithm is the number of neighbors also. With a help of a for loop I've tried different n_neighbour parameters from 1 till 50 to test when the accuracy starts to stagnate. Above 20 the accuracy was not significantly growing, reaching around 0.775 private score.

```
for i in range(50):
  from sklearn.neighbors import KNeighborsClassifier
  model = KNeighborsClassifier(n_neighbors=(i+1), weights='distance')
  model = model.fit(X_train, y_train)
  y_pred = model.predict(X_test)
  print("i: ", i, " F3 score  ",fbeta_score(y_test, y_pred, average='binary', beta=3))
```

```
i:  0  F3 score   0.662033650329188
i:  1  F3 score   0.662033650329188
i:  2  F3 score   0.7387769529869211
i:  3  F3 score   0.7438894792773644
i:  4  F3 score   0.7566248256624826
i:  5  F3 score   0.7592722183344995
i:  6  F3 score   0.7644413697682462
i:  7  F3 score   0.7630939993062781
i:  8  F3 score   0.7673860911270981
i:  9  F3 score   0.767123287671233
i:  10  F3 score   0.7697547683923704
i:  11  F3 score   0.7681849082256968
i:  12  F3 score   0.7791327913279132
i:  13  F3 score   0.777552400270453
i:  14  F3 score   0.782198246797033
i:  15  F3 score   0.782198246797033
i:  16  F3 score   0.7850404312668464
i:  17  F3 score   0.7896505376344086
i:  18  F3 score   0.7870060281312793
i:  19  F3 score   0.7900903916973552
i:  20  F3 score   0.7921122994652406
i:  21  F3 score   0.7977303070761014
i:  22  F3 score   0.7961359093937376
i:  23  F3 score   0.7984031936127745
i:  24  F3 score   0.80371969445367
i:  25  F3 score   0.7990716180371352
i:  26  F3 score   0.7998672419515434
i:  27  F3 score   0.7990716180371352
i:  28  F3 score   0.7998672419515434
i:  29  F3 score   0.8007941760423561
i:  30  F3 score   0.8002645502645502
i:  31  F3 score   0.799207397622193
i:  32  F3 score   0.7966942148760331
```

.
After changing the parameters mentioned above, I've made the best kNN with the selected best parameters and the accuracy was the following:

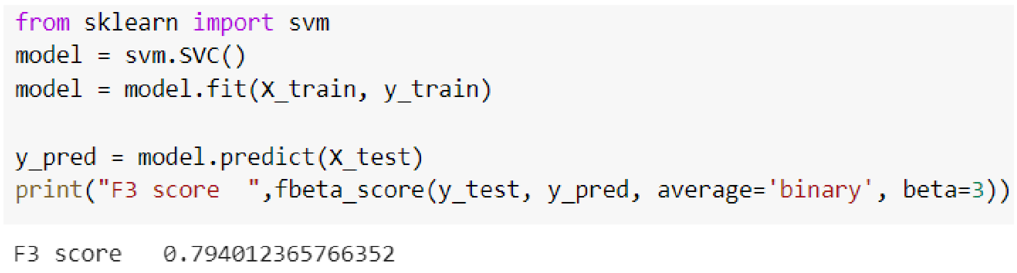
```
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(p = 1, n_neighbors=(24), weights='distance')
model = model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print("F3 score  ",fbeta_score(y_test, y_pred, average='binary', beta=3))
```

```
F3 score   0.8193418401611821
```

Private score: 0.77253
Public score: 0.79189

e. Support vector Machine

This is a type of supervised learning algorithm that can be used for classification or regression tasks. They work by finding the hyperplane in a high-dimensional space that maximally separates the data points of different classes. To find the optimal hyperplane, the SVM algorithm searches for the one that has the greatest margin, or distance, between the closest data points of each class. The data points that are closest to the hyperplane are called support vectors and have a key role in determining the position of the hyperplane.



```python
from sklearn import svm
model = svm.SVC()
model = model.fit(X_train, y_train)

y_pred = model.predict(X_test)
print("F3 score  ",fbeta_score(y_test, y_pred, average='binary', beta=3))
```

```
F3 score   0.794012365766352
```

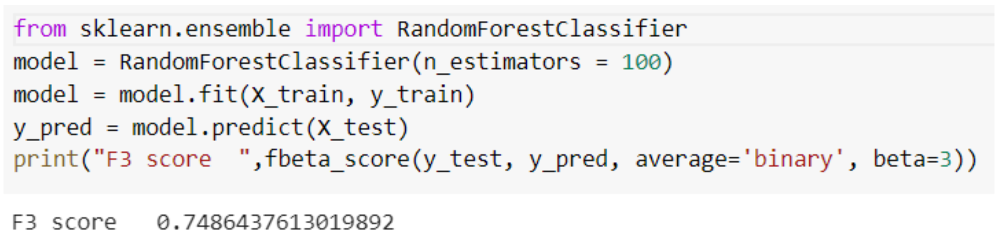
Private score: 0.76923
Public score: 0.77012

8. Ensemble learning

Ensemble learning is a machine learning technique which uses multiple models to obtain better predictive performance than could be obtained from any of the individual models alone. The models are trained using different algorithms and their results are combined to produce the final predictions. This technique is often used in data science competitions and is particularly effective in improving the accuracy of predictive models.

a. Random Forest

This is a combination of multiple machine learning models, specifically decision trees. It is called "random" because it randomly selects a subset of features to use as decision nodes in each tree. Most important parameter in the Random Forest Classifier is the “n_estimators”, which defines the number of trees one would like to train.



```python
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier(n_estimators = 100)
model = model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print("F3 score  ",fbeta_score(y_test, y_pred, average='binary', beta=3))
```

```
F3 score   0.7486437613019892
```

Parameters	**Private score**	**Public score**
n_estimators = 10	0.70267	0.73517
n_estimators = 100	0.77501	0.77676
n_estimators = 500	0.78285	0.79508

n_estimators = 500
criterion = ‘entropy’
	0.80727	0.82315

n_estimators = 500
criterion = ‘entropy’
bootstrap = False
	0.80393	0.795

n_estimators = 500
criterion = ‘entropy’
bootstrap = True
	0.81494	0.82103

b. Boosting

Boosting explicitly seeks models that complement one another. Boostings have many parameters that can be tuned in order to make more accurate classification.

i. **AdaBoost**: Private score: 0.78025, Public score: 0.75956
The weak models in AdaBoost are typically decision trees with a single split, known as "decision stumps." The idea behind AdaBoost is to train the weak models in a sequence, where each model is trained to correct the mistakes of the previous model.

ii. **XBG** Private score: 0.80748, Public score: 0.82116
A high-performance implementation of gradient boosting, with additional features such as parallelization, regularization, and handling of missing values.

```
from xgboost import XGBClassifier
model = XGBClassifier()
model = model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print("F3 score  ",fbeta_score(y_test, y_pred, average='binary', beta=3))
```

```
F3 score   0.8229342327150085
```

iii. **Gradient Boosting** Private score: 0.80021, Public score: 0.80168
A prediction model in the form of an ensemble of weak prediction models, typically decision trees. It builds the model in a stage-wise fashion like other boosting methods do, and it generalizes them by allowing optimization of an arbitrary differentiable loss function.

```
from sklearn.ensemble import GradientBoostingClassifier
model = GradientBoostingClassifier()
model = model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print("F3 score  ",fbeta_score(y_test, y_pred, average='binary', beta=3))
```

```
F3 score   0.807743658210948
```

9. Conclusion and Plans

Preprocessing has a huge effect on the training accuracy and it is crucial to handle outliers and missing values properly. In the future with over and under sampling, preprocessing could make more improvement. It would be also interesting to test other methods in dealing with missing values. It was also very important to focus on the evaluation scoring method and since the data is very imbalanced, without using F-score it was not possible to

calculate accuracy locally.
Parameter tuning has also a very large effect on the models’ performance. I haven't found the best parameters for the algorithms, but doing it would increase classification accuracy.
The basic algorithms are not able to achieve outstandingly high accuracies, but combining them and implementing ensemble algorithms could give the best classifications. I’ve applied boosting and random forest, but improvement could be achieved by using more complex ensemble models, stacking, bagging, and example specific combinations.
In order to create a real world solution I have to take the fact under consideration that false negative prediction has a larger negative consequence therefore maybe a cost function could be implemented and calculated for every classification.