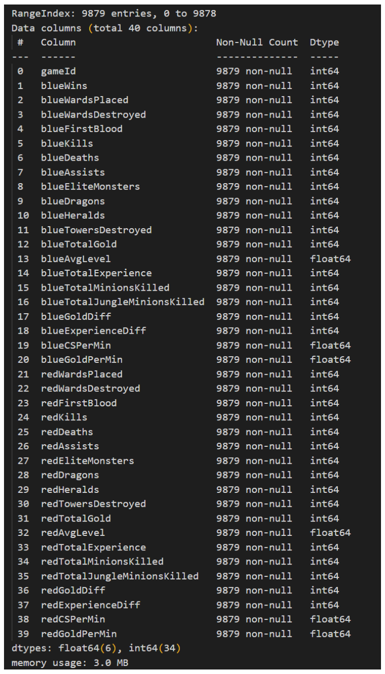
Predicting wins in high ELO league of legends

Antonio Kristófer Salvador
Axel Magnússon

Figure 1: *League of Legends*

1 The dataset

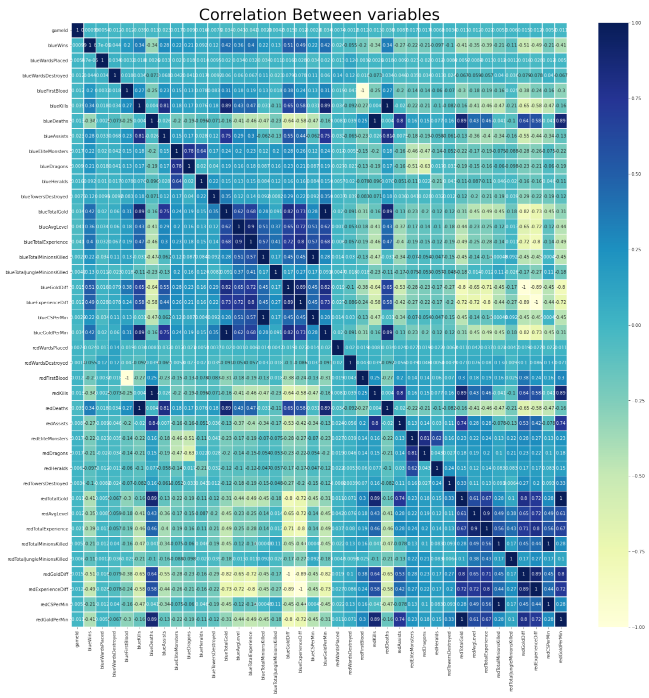
League of Legends is a 2009 multiplayer online battle arena video game developed and published by Riot Games where two teams (red and blue) of 5 players fight to destroy their opponents base. The dataset contains the stats at the ten minute mark for close to ten thousand games from high ELO (ranks from diamond 1 to master) - approximately the top 0.3 percent of players. Our goal in this project is to use these statistics to predict the end result of the game. There are 19 features per team, a unique gameId and the target value blueWins. The dataset comes from here and there are some more details along with a short glossary on some league of legends terms the reader may find useful.



```
RangeIndex: 9879 entries, 0 to 9878
Data columns (total 40 columns):
 #   Column                        Non-Null Count  Dtype
---  ------                        --------------  -----
 0   gameId                        9879 non-null   int64
 1   blueWins                      9879 non-null   int64
 2   blueWardsPlaced               9879 non-null   int64
 3   blueWardsDestroyed            9879 non-null   int64
 4   blueFirstBlood                9879 non-null   int64
 5   blueKills                     9879 non-null   int64
 6   blueDeaths                    9879 non-null   int64
 7   blueAssists                   9879 non-null   int64
 8   blueEliteMonsters             9879 non-null   int64
 9   blueDragons                   9879 non-null   int64
 10  blueHeralds                   9879 non-null   int64
 11  blueTowersDestroyed           9879 non-null   int64
 12  blueTotalGold                 9879 non-null   int64
 13  blueAvgLevel                  9879 non-null   float64
 14  blueTotalExperience           9879 non-null   int64
 15  blueTotalMinionsKilled        9879 non-null   int64
 16  blueTotalJungleMinionsKilled  9879 non-null   int64
 17  blueGoldDiff                  9879 non-null   int64
 18  blueExperienceDiff            9879 non-null   int64
 19  blueCSPerMin                  9879 non-null   float64
 20  blueGoldPerMin                9879 non-null   float64
 21  redWardsPlaced                9879 non-null   int64
 22  redWardsDestroyed             9879 non-null   int64
 23  redFirstBlood                 9879 non-null   int64
 24  redKills                      9879 non-null   int64
 25  redDeaths                     9879 non-null   int64
 26  redAssists                    9879 non-null   int64
 27  redEliteMonsters              9879 non-null   int64
 28  redDragons                    9879 non-null   int64
 29  redHeralds                    9879 non-null   int64
 30  redTowersDestroyed            9879 non-null   int64
 31  redTotalGold                  9879 non-null   int64
 32  redAvgLevel                   9879 non-null   float64
 33  redTotalExperience            9879 non-null   int64
 34  redTotalMinionsKilled         9879 non-null   int64
 35  redTotalJungleMinionsKilled   9879 non-null   int64
 36  redGoldDiff                   9879 non-null   int64
 37  redExperienceDiff             9879 non-null   int64
 38  redCSPerMin                   9879 non-null   float64
 39  redGoldPerMin                 9879 non-null   float64
dtypes: float64(6), int64(34)
memory usage: 3.0 MB
```

2 Preprocessing

To begin with preprocessing we first looked to see if any values were null or otherwise invalid, we then took note of the correlation between each feature and the target feature.



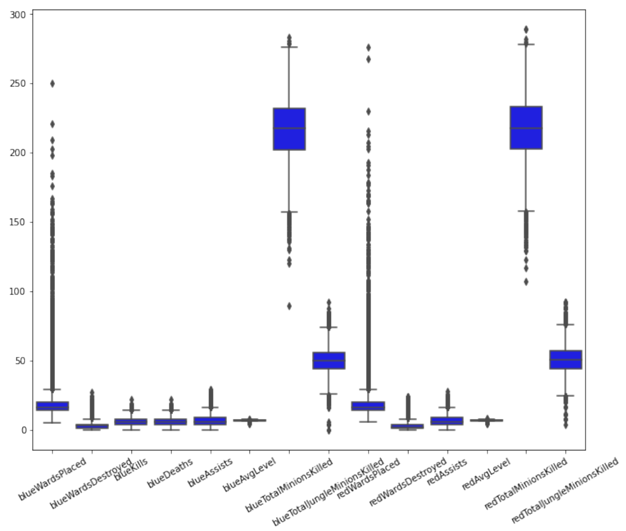
A few features were obviously redundant, having a correlation of one with other features, those features being blueTotalGold and blueGoldPerMin because all games are exactly 10 minutes in and therefore gold/min is simply totalgold/10 same for totalminionskilled and cs/minute (cs stands for creep score - the creeps being minions). So we removed those time based attributes, along with the gameId as it had no correlations.

The feature ”wards placed”, had us confused, the values being far too high, we suspect this column is incorrectly labelled and refers to vision score, which measures effectiveness of wards placed, and destroyed (+1 for each minute of ward lifetime gained and +1 for each minute of ward lifetime denied), however in the end we decided to keep it. both sides have a gold and exp diff column, which are inverses of each other and so we removed the diff attributes from the red side.

The values with the highest correlations with our target value were: Golddiff, Expdiff, Totalgold, Totalexp, Avglvl, Kills and Deaths.

We also removed the attributes redKills and redDeaths because the red teams kills are represented in the blue teams deaths and vice versa.

We tried removing the outliers by filtering out all values outside of three standard deviations, but later decided against it since the outliers are based on two cases: One team is dominating and should therefore be likely to win or one team or some members are deliberately losing, this can be seen in wardsplaced since the only way to have scores like the 250+ ones we observed is to sit in the teams base and use up all of your gold on wards.



We also tried making delta variables for all variables that had both blue and red counterparts but in the end we decided against it since we felt we were losing some information along the way, for example a team leading 14-10 in kills is not as significant as a team leading 4-0 but both would give us a delta value of 4.

3 Models

After preprocessing we decided on which models to try. These are the models we chose: DecisionTreeClassifier, SelectKBest, MLPClassifier, SVC, RandomForestClassifier, LogisticRegression, KNeighborsClassifier

DecicionTreeClassifier: Because we used this in a lab assignment and were somewhat familiar with it. We also did a second DecicionTreeClassifier but using SelectKBest for comparison.

We tried the MLPClassifier because we wanted to try working with neural networks.

Logistic Regression: This algorithm is used for binary classification problems which is exactly what we have so we thought this could work well.

SVC: Works well with 2 classes like logistic regression, is easy to use and unlike decicion trees they tend to not overfit data.

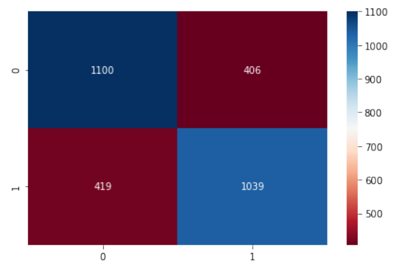
All of our models were trained using GridSearchCV in conjunction with sklearn pipelines, this allowed us to easily test many permutations of different hyperparameters for each model efficiently, while providing cross validation simultaneously.

4 Results

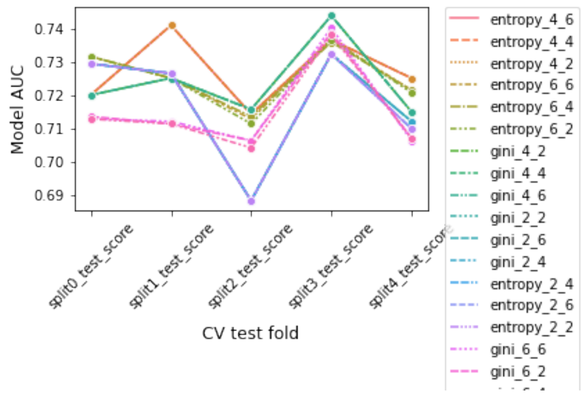
4.1 introduction

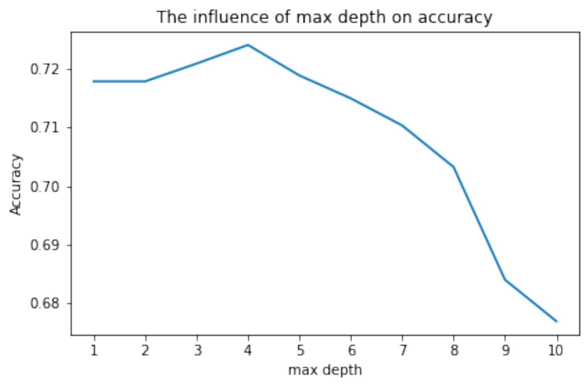
We include a confusion matrix and a classification report for the best model of each type.

4.2 Decision tree



```
              precision    recall  f1-score   support
 blue loses        0.72      0.73      0.73      1506
 blue wins         0.72      0.71      0.72      1458
-------------------------------------------------------
 accuracy                              0.72      2964
 macro avg         0.72      0.72      0.72      2964
 weighted avg      0.72      0.72      0.72      2964
```





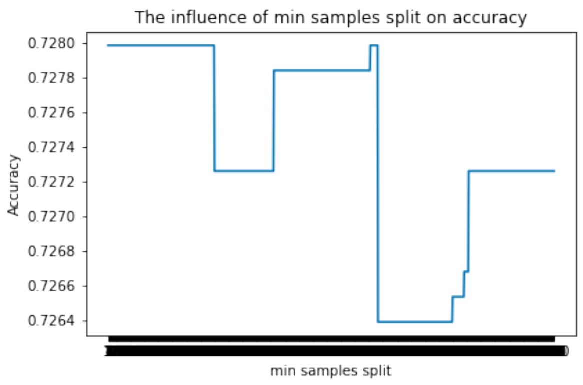
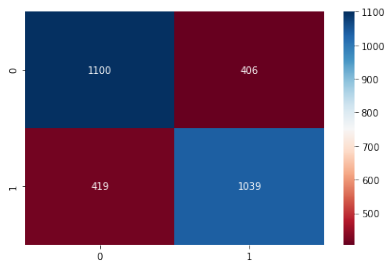


Figure 2: *min samples split ranging from 2 - 1000*

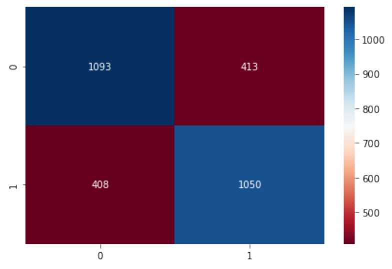
4.3 SelectKBest



```
                precision    recall  f1-score   support
  blue loses       0.72       0.73      0.73      1506
   blue wins       0.72       0.71      0.72      1458
---------------------------------------------------------
    accuracy                            0.72      2964
   macro avg       0.72       0.72      0.72      2964
weighted avg       0.72       0.72      0.72      2964
```

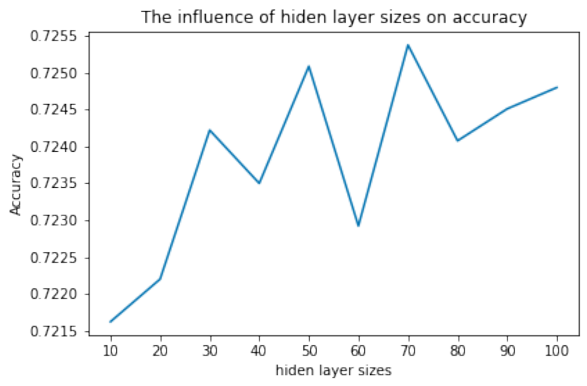
The K value had very little effect, results were largely similar to the regular decision tree.

4.4 MLPClassifier

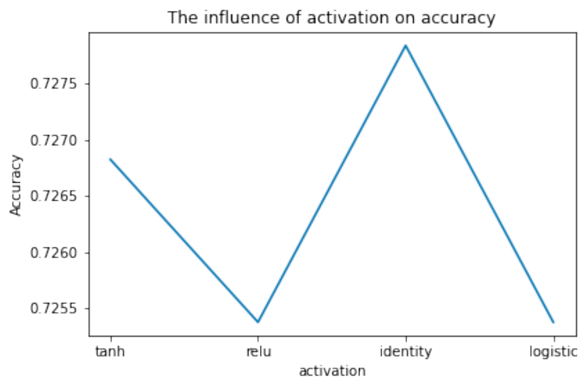


```
                precision    recall  f1-score   support
  blue loses       0.73       0.73      0.73      1506
   blue wins       0.72       0.72      0.72      1458
---------------------------------------------------------
    accuracy                            0.72      2964
   macro avg       0.72       0.72      0.72      2964
weighted avg       0.72       0.72      0.72      2964
```

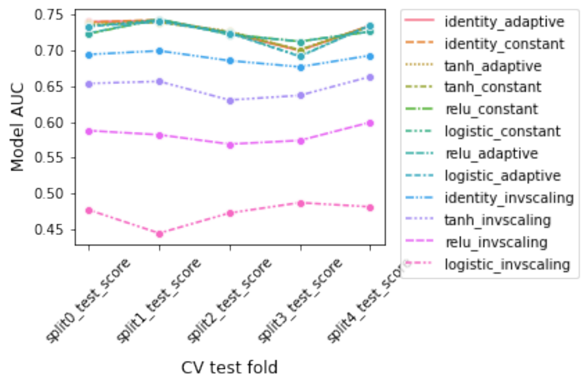
The hidden layer sizes were rather confusing, we believe we hit somewhat of a sweetspot at around (70,) however this may not be the best value, we see a peak at (30,) which corresponds with our number of attributes, the greater values may simply be approximating more complex networks in a single layer, we also tested multi layer setups, but failed to find one that performed better than their singly layered counterparts.



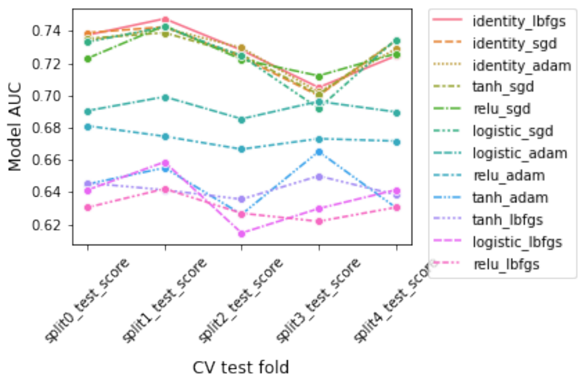
Identity was not as consistently the best performing activation function as this graph depicts, the rankings seemed to vary based on the hidden layer sizes. However relu and logistic were consistently the worst.



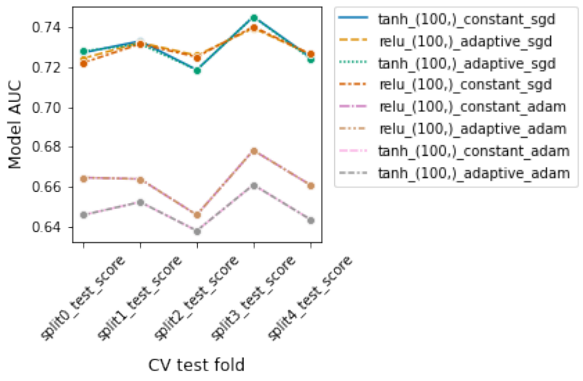
As for learning rates invscaling was consistently the worst performing, but the others were mixed.



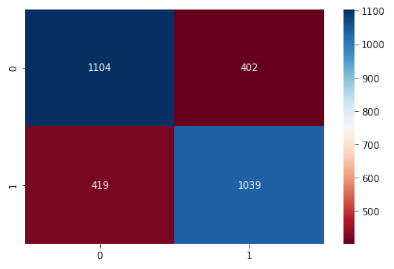
Solver: sgd was best most of the time, except when using the identity activation.



This next graph shows how the performance of different activation functions relied more on the other values as the ordering is inconsistent. It also shows how sgd was consistently better than adam.

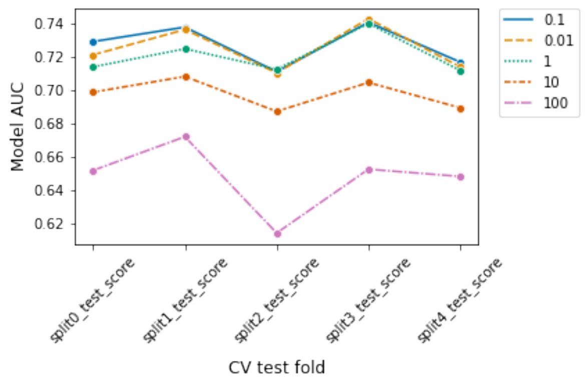


4.5 SVC

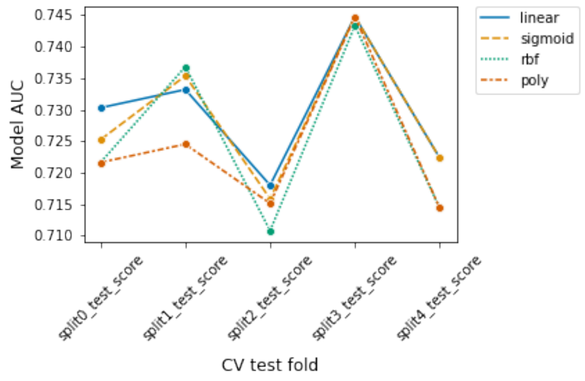


```
                 precision      recall    f1-score     support
  blue  loses         0.72        0.73        0.73        1506
   blue  wins         0.72        0.71        0.72        1458
----------------------------------------------------------------
     accuracy                                 0.72        2964
    macro  avg        0.72        0.72        0.72        2964
 weighted  avg        0.72        0.72        0.72        2964
```

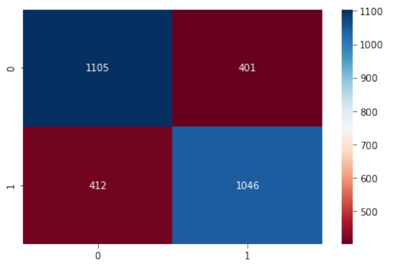
Smaller C values consistently outperformed larger ones, smaller values here being values lower than one. We believe this is because the dataset is rather complex but still well separable if we give the model enough freedom. We also tested extremely small values and this trend continued, the best performing tested value being 0.001, which makes sense as the data is quite noisy, since many games are either not decided yet at ten minutes or a team may blow their lead and lose.



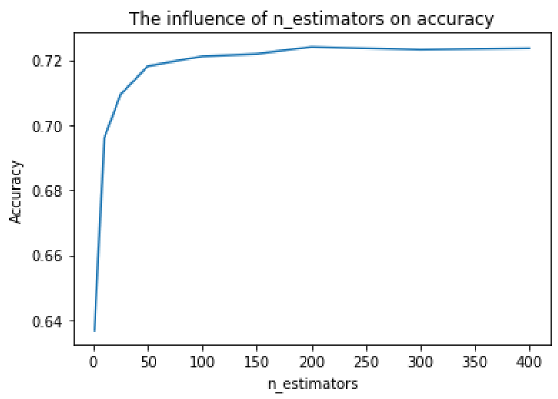
The kernels were somewhat similar in performance, but linear and sigmoid were on top for most tests. This leads us to believe our dataset is at least somewhat linearly separable.



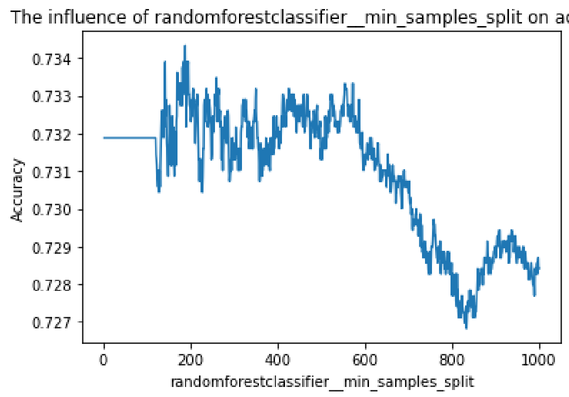
4.6 Random Forest Classifier



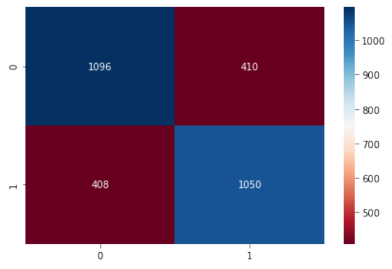
```
                 precision      recall    f1-score     support
  blue  loses         0.73        0.73        0.73        1506
   blue  wins         0.72        0.72        0.72        1458
----------------------------------------------------------------
     accuracy                                 0.73        2964
    macro  avg        0.73        0.73        0.73        2964
 weighted  avg        0.73        0.73        0.73        2964
```



This graph shows the influence of the number of trees in the forest. We tested values ranging from 1-400 and as we can see we the graph stabilizes and we start to have diminishing returns at approximately n=100.



4.7 Logistic Regression



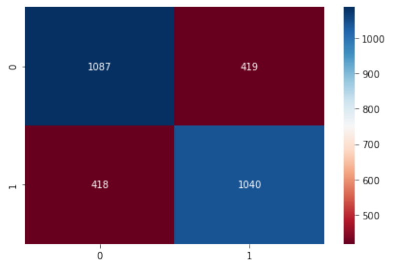
```
              precision    recall  f1-score   support
  blue loses       0.73      0.73      0.73      1506
   blue wins       0.72      0.72      0.72      1458

-----------------------------------------------------
    accuracy                           0.72      2964
   macro avg       0.72      0.72      0.72      2964
weighted avg       0.72      0.72      0.72      2964
```

For logistic regression we tried changing the solver algorithm but this ended up having negligible effect on accuracy so we stuck with the default ”lbfgs”.

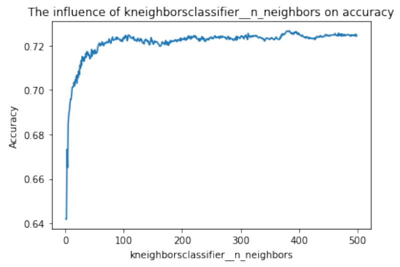
4.8 KNeighborsClassifier

This graph is almost identical to the one in the RFC. We see a big jump in accuracy gain from 1-100 but then it stabilizes.



```
              precision    recall  f1-score   support
  blue loses       0.72      0.72      0.72      1506
   blue wins       0.71      0.71      0.71      1458

-----------------------------------------------------
    accuracy                           0.72      2964
   macro avg       0.72      0.72      0.72      2964
weighted avg       0.72      0.72      0.72      2964
```



5 Conclusions

All of our models ended up having roughly the same accuracy, around 0.71 - 0.73. We believe this is close to the limit we can reach with the training data we are working with. The model which was selected was not always of the same type, we believe this is a result of all of the models being so close in performance and the best one decides on random factors.

6 Future work

We might have achieved a better result by focusing more of our efforts on the feature engineering side, we would spend more time on this next time. We should also set the random state attribute on all models to have more concrete performance comparisons.