Word count: 1379

CE802 Report

From my initial pilot study, I selected that for the classification problem which was described I would use decision trees to acquire the best classification results. From the experiments. During conducting the experiments in part 2 and 3 where I was required to use a combination of classification and regression methods, I found that my initial hypothesis was wrong, which I will explain further here. During each experiment I selected three different of the appropriate algorithms (either classification or regression). I ran each of these algorithms twice, the first time was with no parameter tuning this was done to get a baseline result. I then ran them a second time performing hyperparameter tuning on each of the models. This allowed me to improve the baseline score to get the optimal values for each model.

The first task was a classification problem. The methods I used for this where Decision tree, KNN and Support vector classifier. These are all methods I spoke about in my pilot study where I concluded saying that a decision tree would be best. When initially using the DT (Decision tree) classifier It returned an accuracy of 0.78. To tune the algorithm, I created a parameter grid to apply different parameters to the algorithm. This grid is a dictionary which is used when hyper tuning to find the optimal parameters for a given algorithm. For the DT the parameters I changed was:

min_sample_split: ranged from 1 to 50

criterion: Gini or Entropy

splitter: best or random

When iterating over these parameters I found that I was able to increase the accuracy to 0.80, which is only a small 2% increase. When compared to the initial run. I found that the best parameters where *{'criterion': 'entropy', 'min_samples_split': 32, 'splitter': 'random'}* and that this was reached on the 159th run. The results can be observed in figure 1. Showing the mean test score across all 198 runs of the program.

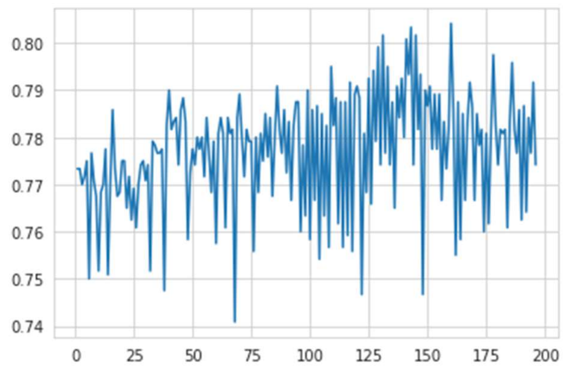

Figure 1 – Generated with matplotlib

The second algorithm I used was a KNN classifier. I went through the same process with this algorithm as I did above. First running it using its default parameters, then with hyper tuning. The initial run of this algorithm gave an accuracy of 0.74, which is lower than that of the initial DT run. The second run of this algorithm required parameter tuning. The parameters I changed was:

n_neighbours: ranged from 1 to 30

weights: uniform or distance

algorithm: kd_tree, ball_tree or brute

When iterating over these parameters I was able to equal the accuracy of the tuned DT with this algorithm reaching an accuracy of 0.80. However, this is a larger increase than before at 6%. The best parameters I found where *{'algorithm': 'kd_tree', 'n_neighbors': 28, 'weights': 'distance'}.* When comparing this to the DT they both have the same accuracy. However, the runtime of the KNN algorithm was much higher than that of a DT even though they both had a similar number of parameter combinations to iterate over. Therefore, like I said in my initial hypothesis out of these two algorithms I would use a DT. Below, in figure 2 you can see the mean test score for the iterations of the program. This shows something interesting. It can be observed that changing the algorithm to computer the nearest neighbour, in this case does not affect the result, you can visibly see that all three algorithms follow a very similar pattern.

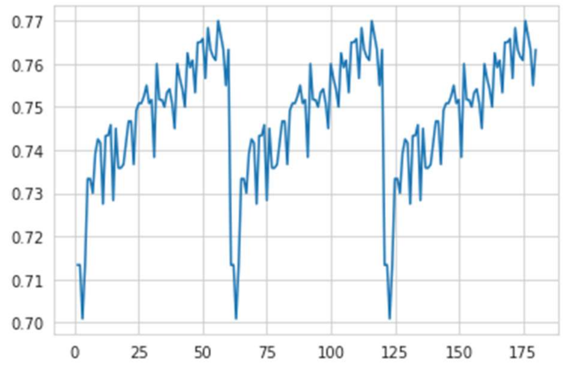

Figure 2- Generated with matplotlib

The final algorithm I used for the classification task was a support vector classifier. Again, I used the same testing as before. From the start it seemed this algorithm was going to be strongest. With the initial run yielding a accuracy of 0.82. Which is already higher than both the tuned DT and knn algorithms. When tuning the algorithm, I changed the following parameters:

C: 1, 10, 100, 1000

Gamma: 1, 0.1, 0.001, 0.0001

Kernel: poly or rbf

When iterating through these parameters I was able to greatly increase the accuracy to 0.91. increasing it by 9%. The best parameters I found where *{'C': 100, 'gamma': 0.1, 'kernel': 'poly'}*. This made the support vector classifier by far the most accurate of the three algorithms. However, the time to execute the code was much longer than all the other methods. This is what I hypothesised in my study pilot. However, it is still the most accurate and therefore the best algorithm to use. The main parameter we changed here was C. This is what balances the penalty and margin width within the SVM. This helps us increase the accuracy in the algorithm more than the others as helps us handle outliers within the dataset. When C is smaller the outliers are ignored, and the vector margin is a lot wider. When C is larger the outliers will still be counted in the results but will have a penalty associated with it. We can see that in our optimal parameters C was set to 100. This means that a medium penalty was given to the outliers. Not as large as it was at 1000 but they are also less ignored than when it was set to either 1 or 10. The kernels I selected to test for this was polynomial and Gaussian radial basis function. I also tried to include a linear kernel when testing. However, this massively increased the runtime and did not provide an increased accuracy and therefore was removed.

When looking at the regression problems I carried out the study the same way. As I couldn't use the same accuracy metric to measure the algorithms, I instead looked at the r2 score and the mean test score when tuning the algorithms. The first algorithm was linear regression. Both the initial run of this algorithm and the tuned version gave the same mean squared error and r2 score of 0.79. This shows that the default parameters for this are the most accurate for it. These parameters where (*copy_X=True, fit_intercept=True, n_jobs=None, normalize=False).* In the graph below of the mean test score you can see that when the fit intercept parameter is changed to False the score drops to 0.

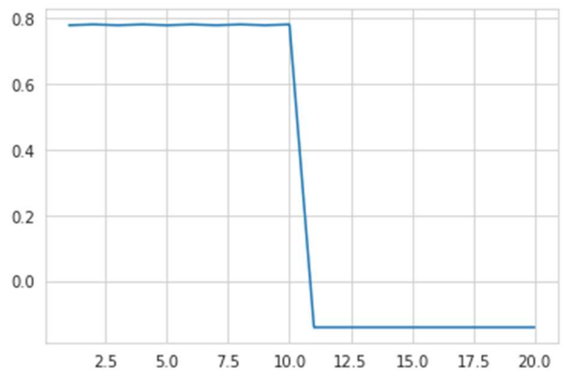

Figure 3- Generated with matplotlib

The next algorithm I used was support vector regression. As this gave the highest accuracy in the classification task, I was interested to see how it would compare in a regression task. To make the comparison fair I used the same parameters when tuning it except for the kernel in which I used linear instead of polynomial. The initial run gave an r2 score of -0.2. This is likely because C is set to 1 meaning that the outliers are mostly ignored, which isn't very helpful. When tuning it the r2 score increased to 0.87. Much higher than the initial run. When looking at the parameters used to get this you can see that C was increased to 1000 meaning that the penalty given was much higher, making the outliers more useful to us as compared to completely ignoring them.

The final algorithm I used was gradient boosting regression. When initially running this with the default parameters it reached a r2 score of 0.85. When tuning it the only parameter which was changed was the estimators. What I found from doing this was the higher the number of estimators the more accurate the algorithm was This is because this algorithm is robust with overfitting, so it does not affect the outcome as much. When doing this I was able to reach an r2 score of 0.86 only a slight increase on before. From figure 4 below you can see that the more estimators is better, but when it gets to 400 it starts to balance of meaning that increasing it anymore would not yield a better result.

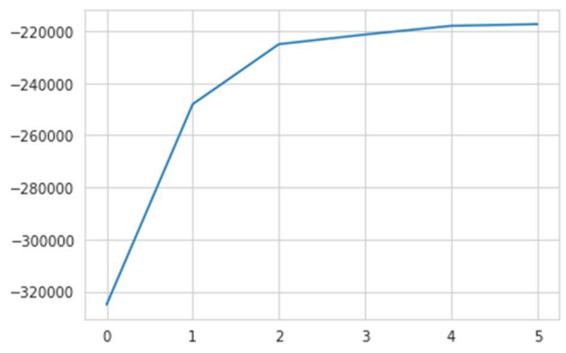

Figure 4 - Generated with matplotlib

From the study above the results show that the best algorithm for both the regression and classification task from the ones which I tested is the SVR and SVC. However, the parameters used where different. In the regression task a rbf kernel was used instead of the polynomial one in the classification task. The regression task also had a much higher C value at 1000, applying a larger penalty to the values outside the margin.