**Machine Learning - Final Assignment – Killian Ronan – 18328687**

1. I chose two stations which should have different patterns of behaviour. Firstly, I chose to analyse the Pearse Street bike station for its central location in the city. Next, I chose Avondale Road bike station for its suburban location in Phibsborough, north Dublin.

**Feature Engineering:**

After downloading the dataset provided, I wrote a function which created the features I wished to used by parsing them from the dataset. To begin feature selection process I thought it would be a good idea to create a linear model and target the coefficients of the first few features I decided to use. The initial features I included were the minute of the hour, hour of the day, day of the month and the month using the timestamp of each row. The dataset given contained some columns with constants which would not be useful as features which I did not include such as geographic coordinates for the bike station, addresses and station ID’s.

Using the lecture notes as guidance, I thought that a feedback model approach would work appropriately for this analysis. The previous predictions which we use as features become more important for larger q steps. I gathered the previous three available number of bikes for each timestamp and used these as features for when training the model initially. These previous number of bikes features will later be replaced by the predictions between the target prediction and the target prediction minus the step size. I passed these features to a linear model and looked at the weights of the coefficient features. I found that the coefficient weights for the ‘month of year’ and ‘minute of hour’ features were the smallest. This showed they were having the least effect on the data compared to the other features. I decided to fit the training data to a Lasso model, because it can will shrink less important features towards zero. This confirmed which features were the unimportant ones. I printed the coefficient values and found the features(‘month of year’ and ‘minute of hour’ ) were shrunk to 0. The coefficient features which were shrunk to zero can be seen printed below.



For this reason, I decided to remove them to see how it would affect the performance of my models. I saw little of impact on performance after removing them, but I thought the ‘minutes in hour’ feature may hold more importance and shouldn’t be discarded. I tried combining the hour and minute features into one feature which was an integer representing the two. I found that it still kept a significant weight once fitted to the model so I kept it.

The highest weight was on the previous prediction with the 2 previous to that being weighted less. However, when I started to make predictions my model wasn’t performing as well as I thought it could with a low accuracy. I decided I needed more features and thought that adding the number of bikes at certain time intervals before each timestamp might help improve performance. I added features for the number of bikes one hour, one day and one week before the current timestamp. Each row from the dataset increments in 5 minute intervals, so these values were easy to locate. After adding these features I ran my model which began to perform a lot better. I looked at all coefficient feature weights and found they all did not shrink to 0 when fitting a Lasso model, so I kept them in my final solution.

The final features along with example values can be seen in the table below (x\_train):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| hour | previousBike1 | previousBike2 | previousBike3 | weekBeforeBikes | dayBeforeBikes | hourBeforeBikes |
| 9 | 54 | 53 | 54 | 58 | 49 | 51 |

|  |
| --- |
| noBikes |
| 50 |

The output column can be seen here (x\_test):

The next step was to decide on which methodology I would use to tackle this problem.

**Methodology:**

There are around 20,000 rows of data for each bike station I used. I decided that using a sliding window architecture would suit the models chosen. This would help reduce the training time as it would be too costly if the whole set of data was used for training. The data from up to a week before the target timestamp was used as training data. The sliding window is described well in the slides on time series.

Chart, scatter chart

Description automatically generated

I decided to use a linear based model and a more complex model. I chose to use Lasso Regression and K Nearest Neighbours. I chose Lasso regression for its simplicity, being a linear mode, and also because it usually holds less variance than other linear models. I chose K Nearest Neighbours as it differs from Lasso regression a lot, and it is a method which can and has been used to solve time series problems in the past. I did not want to choose two similar models such as Ridge and Lasso as the results may be too similar for an interesting comparison. The stark differences in the linear Lasso model and non-linear and also non-parametric KNN model, provides me with two very different approaches to solving this problem and it will be interesting to see how they both perform.

The C values for both models were determined using K-Fold cross validation, taking into account the mean squared error and accuracy scores. I thought these performance metrics would work best as, things like F1 score, precision and accuracy would be tougher to obtain as there are only true or false predictions obtained based on whether the prediction is equal to the target or not. I used a random baseline classifier to compare the chosen models against. It had very little accuracy but gives a good basis to compare against. I implemented a feedback models for both KNN and Lasso Regression.

To calculate the final performance of both models, after choosing optimum C and K values, I calculated the accuracy of the model as well as the average miss size for incorrect predictions. I will use these values to compare the performance of both models for both bike stations. Finally I will conclude whether it is feasible to predict the number of bikes in each station 10, 30 and 60 minutes into the future.

**Lasso Regression:**

Lasso regression is a form of regularized linear regression. The strength of the regularisation is determined by the parameter alpha parameter: α = 1/2C. Lasso regression uses a method called shrinkage, in which the coefficients features are shrunk towards 0 which can be used to eliminate unimportant features. It uses linear regression to obtain these coefficients and, lasso regression to regularize them which can help with overfitting. It uses L1 regularization which penalises using the absolute value of the magnitude of coefficients. For larger C values, the L1 penalty increases which results in more coefficient values shrinking closer to zero. This can help in producing simpler models than most, which I thought to be ideal for this problem. As Lasso uses a form of linear regression, I also looked at augmenting the features using polynomials to see its impact on predictions.

**K Nearest Neighbours:**

K Nearest Neighbours (KNN) is an algorithm which uses the distance between test data and training data points to prediction the correct class. It assumes that similar things exist near to one another. The value of K determines the number of neighbour nodes which are compared within the feature space. The prediction is made by choosing the class which is most similar in this list of K nodes. Euclidean distances between a feature vector and each training point are calculated of the K number of neighbours. Each neighbouring point has a weight which is a function of its distance from the input. This means that closer neighbours have a higher contribution to the average than further neighbours. There are different ways of implementing KNN, one such uses Gaussian weighting. Choosing too high or too low a value for K can result in overfitting or underfitting. We can use cross validation to determine the optimal K value. KNN is more complex than Lasso regression and takes more time to run due to its computational complexity as it needs to search the training data to find the closest neighbours.

**Evaluation:**

**Random Baseline:**

The results obtained for each step size using the random baseline classifier can be found in the table below. The results are extremely poor but can be used a basis to compare my other models against. Otherwise this predictor is next to useless.

|  |  |  |
| --- | --- | --- |
| **Avondale Road Station** | | |
| **qStep** | **Accuracy** | **Average Miss Size** |
| **2 (10 minutes)** | 3.582% | 10.0006 |
| **6 (30 minutes)** | 3.412% | 10.1722 |
| **12 (60 minutes)** | 3.801% | 10.0031 |

|  |  |  |
| --- | --- | --- |
| **Pearse Street Station** | | |
| **qStep** | **Accuracy** | **Average Miss Size** |
| **2 (10 minutes)** | 3.648% | 9.1146 |
| **6 (30 minutes)** | 3.555% | 9.238 |
| **12 (60 minutes)** | 3.429% | 9.3247 |

**Lasso Regression:**

To begin, I performed 5-fold cross validation on both datasets to determine what the optimum C value was to use. The C value indicates the size of the regularization penalty used. For lower values of C, we can expect that less coefficient features will be shrunk towards 0 as the penalty is lower. For higher C values we can expect the opposite, that more coefficient features will be shrunk towards 0 as the penalty is higher. I plotted the mean squared error graph, using the mean and standard deviation of the error rate, for a range of C values (0.001, 0.01, 0.1, 1, 10, 20) which can be seen below.

**Avondale Station** **Pearse Station**

Chart

Description automatically generated Chart

Description automatically generated

We can see here that for values of C above 0.1/1, the mean squared error does not change. This indicates that a lower C value may be optimal, below 1 to avoid overfitting the data. We also want to avoid underfitting, and we can see that occurring when C has a low value around 0.001. For C = 0.001 the mean squared error increases immensely compared to other C values. Using these observations we can estimate that a C value of around 0.1/1 might work best. Next we can take into account the accuracy scores returned from the cross validation. Once again, I graphed the scores for accuracy against the different C Values and included the graph below.

**Avondale Station** **Pearse Station**

Chart

Description automatically generated Chart, box and whisker chart

Description automatically generated

Here we can see the accuracy is quite low for lower values of C, as the penalty is higher due to the alpha parameter for Lasso regression being 1/2\*C. This means for values of C below 1 but above 0, the penalty is larger than using higher C values. The accuracy score stays around the same for values of C above 1. We want to avoid overfitting, and we want to take into account our observations from the mean squared error graph where we decided that a C value of around 0.1 or 1 would be best. This accuracy graph further backs an optimal C value of 1, so I will choose this. I also looked at augmenting the features using polynomials of degree 3. I did not increase this any higher as it meant the run time was just too long to investigate. The final accuracy using the augmented features was slightly lower than that of using the original features, so I decided against using them. The graphs for both stations were very similar, so the chosen C value seemed to be optimal for both of them.

The values I obtained for each required step size were printed to the console in the attached code and can be seen in the tables below.

|  |  |  |
| --- | --- | --- |
| **Avondale Road Station** | | |
| **qStep** | **Accuracy** | **Average Miss Size** |
| **2 (10 minutes)** | 87.147% | 0.1694 |
| **6 (30 minutes)** | 29.676% | 1.6714 |
| **12 (60 minutes)** | 29.391% | 1.6951 |

|  |  |  |
| --- | --- | --- |
| **Pearse Street Station** | | |
| **qStep** | **Accuracy** | **Average Miss Size** |
| **2 (10 minutes)** | 71.286% | 0.4937 |
| **6 (30 minutes)** | 19.454% | 2.9829 |
| **12 (60 minutes)** | 19.503% | 3.0032 |

We can see that for higher step sizes, the accuracy decreases and the average miss size increases. This is expected as we are predicting further into the future, which is a harder task to perform as there is more room for error or fluctuation in number of bikes. The model performs excellently for the smallest step size, 10 minutes into the future, predicting correctly almost 9/10 times for the Avondale station and just under 7/10 times for Pearse station. The miss size on average is also very small, the highest being 3 with Pearse station 60 minutes into the future and the lowest being Avondale station 10 minutes into the future with 0.1694. These observations indicate the choice of using Lasso Regression was a good one as the performance was high. The accuracy of the model using the Pearse station data is lower than the Avondale station. The average miss size is also larger for Pearse station. This model outperformed the random baseline classifier immensely, as expected due to the extremely poor performance of the baseline.

**K Nearest Neighbours:**

Once again, I began by performing 5-fold cross validation on both datasets to determine what the optimum K value was to use. KNN attempts to predict the right class by measuring the distance between test data points and training data points. The K value indicates the number of neighbours to use. For lower values of K we can expect overfitting to occur, as there are not enough neighbours taken into account for predictions. Conversely, for higher values of K we can expect underfitting to occur. Once again, I plotted the mean squared error graph, using the mean and standard deviation of the error rate, for a range of K values (1, 2, 3, 4, 5, 6) which can be seen below.

**Avondale Station** **Pearse Station**

Chart, line chart

Description automatically generatedChart, line chart

Description automatically generated

The results seen here are not amazing, as the mean squared error seems to increase as we increase K up to K = 3. It then seems to diminish past this point lowering for 4, 5 and 6 for both stations. We don’t want to choose a value of 1 for K, although the error rate is lower than most values of K so as to avoid underfitting. We also don’t want to choose too high a value. It seems a value of K around 5/6 might suit best for this model based on both graphs. To help us further with determining the best K value, we can take into account the accuracy scores returned from the cross validation. Again, I graphed the scores for accuracy against the different K Values and included the graph below.

**Avondale Station** **Pearse Station**

Chart, line chart

Description automatically generatedChart, line chart

Description automatically generated

The accuracy scores here are a lot lower than those obtained using the Lasso model. It seems that as we increase K towards 3 the accuracy slowly decreases, following the same trend as shown in the mean squared error graphs where the error rate increased as K increased towards 3. After this point however the accuracy begins to gradually increase again. We know we don’t want to underfit the model so choosing a low value for K is not an option, we also want to avoid overfitting so choosing a value of K too high is not an option either. A value for K around 5 might suit best based on these graphs as K = 6 may be too high and cause overfitting. This matches with our observation from the mean squared error graph, where we estimated a value of 5/6 for K would work best. To conclude I will use a value of 5 for K as the optimal value for the reasons stated above. Again, the graphs for both stations were very similar, so the chosen K value seemed to be optimal for both of them. The values I obtained for each required step size using the optimal value of K = 5 can be seen in the table below.

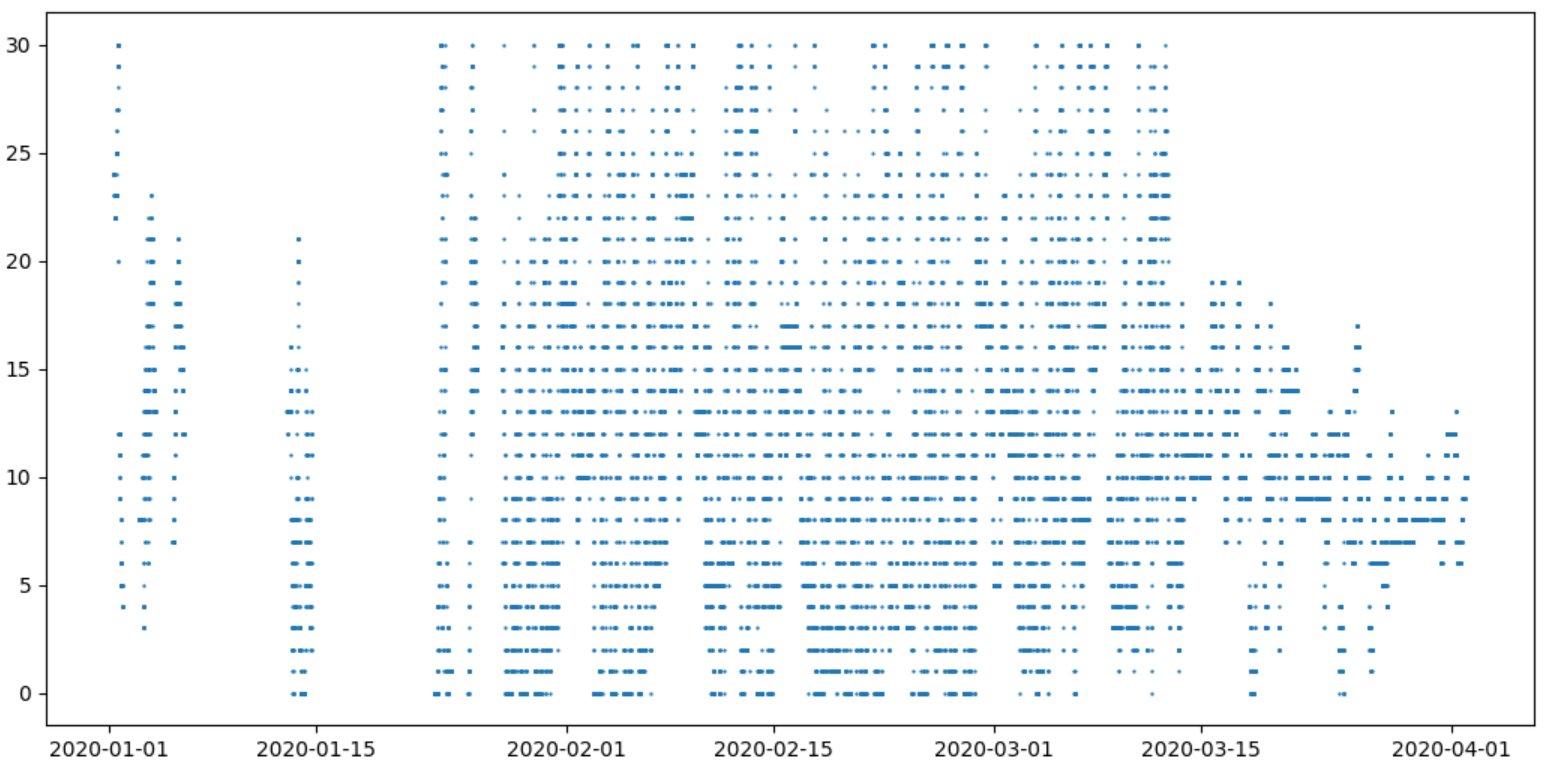
|  |  |  |
| --- | --- | --- |
| **Avondale Road Station** | | |
| **qStep** | **Accuracy** | **Average Miss Size** |
| **2 (10 minutes)** | 44.454% | 1.1924 |
| **6 (30 minutes)** | 38.679% | 2.0848 |
| **12 (60 minutes)** | 32.094% | 2.5479 |

|  |  |  |
| --- | --- | --- |
| **Pearse Street Station** | | |
| **qStep** | **Accuracy** | **Average Miss Size** |
| **2 (10 minutes)** | 34.238% | 1.7362 |
| **6 (30 minutes)** | 25.738% | 3.9609 |
| **12 (60 minutes)** | 21.292% | 4.2128 |

Again, like with the Lasso model we can see that for higher step sizes the accuracy decreases and the average miss size increases. The KNN model underperforms compared to the Lasso model when it comes to the 2 step prediction (10 minutes into the future), with an accuracy of around 44% and average miss size of around 1.2 for the Avondale station. These results were a little disappointing as the Lasso model scored 87% accuracy which is much better than that of KNN. However KNN seems to outperform the Lasso model when it comes to predicting the number of bikes 6 steps (30 minutes) and 12 steps (60 minutes) into the future. For 6 steps into the future it performed with almost 10% better accuracy for Avondale and 7% better for Pearse. For 12 steps into the future it performed 3% better for Avondale and 2% better for Pearse. These are much smaller improvements in performance, but are improvements nonetheless. It seems to work best for 6 step predictions. Another important point to make is that the average miss sizes for KNN seem to be larger than that of Lassos for every step size. This means that some of KNNs predictions are a lot further from the target value. Taking these observations into account, I can conclude that for longer term predictions, KNN works better than Lasso regression. Once again the accuracy is lower and average miss size is higher for the Pearse station dataset. Again, this model outperformed the random baseline classifier by a huge amount, as expected due to the extremely poor performance of the baseline.

A possible reason for this is because Pearse station is located in the city centre, which means there would probably be more people in the area. This may lead to more rapid fluctuations in the number of available bikes. This would in turn might make it more difficult to predict the number of bikes at any given time. Avondale station is located in a suburb, meaning there should be less people around. This may lead to the changes in the number of bikes coming slower, which would explain why the model can predict these values better than the bike station in the city centre. If we look at the scatter plot of the number of bikes against the timestamps, we get the following graphs:

**Pearse Street**



**Avondale Road**

Chart

Description automatically generated

We can see that although the Avondale Road bike station shows fluctuations in the number bikes available during the given time period, the scatter plot for Pearse Street bike station has much steeper and more rapid changes in number of available bikes. This backs up my point that while the bike stations in the city centre will have a larger population in their area, more people will be around to use them meaning there will be more rapid changes in the number of bikes available. Thus, this makes it a little harder to predict for my models resulting in a slightly weakened performance compared to Avondale Road bike station.

In conclusion, based on the results obtained for both the Lasso and K Nearest Neighbours models we can safely say that it is feasible to predict available bikes 10, 30 and 60 minutes into the future. For longer term predictions (30 and 60 minutes), I would recommend the use of the K Nearest Neighbours model as it outperformed the Lasso model. However for shorter term predictions (10 minutes), I would recommend the use of a linear model such as the Lasso model because the performance was brilliant as can be seen in the results. The results from the use of the Avondale Roads bike station data in the suburban area were a little better than that of Pearse Streets bike station data, which I attempted to explain in the paragraph above.

2. What is a ROC curve. How can it be used to evaluate the performance of a  
   classifier.

ROC curves are used to graph the performance of a classification model. It plots the true positive rate against the false positive rate at different classification thresholds. The true positive rate is also the recall performance metric. Classifiers which are discreet and only return one predicted class will result in a single point ROC curve. However, classifiers which are probabilistic can create a curve through thresholding for the probability that each instance belongs each class. Performance level of the classifier can be measured by looking at how close the curve is to the point (0x,1y) i.e. top left of the graph. Comparing the ROC curve of the classifier with a baseline random classifier can be a good judgement on how well the model is performing, as the random baseline classifier should lie along the diagonal of the graph.

Once the ROC curve has been draw, the AUC (Area Under Curve) can be calculated. The AUC is the 2D area beneath the ROC curve, from (0x,0y) to (1x,1y) because the ROC curve is a probability curve. The AUC can tell us how well the model can distinguish its classes. It can be a good measure of the predictive accuracy of a classifier. Models with larger AUC are generally better at distinguishing between both positive and negative classes.

1. Give two examples of situations when a linear regression would give inaccurate

predictions. Explain your reasoning.

Linear regression is the most trivial of machine learning models we have studied. It is a linear model which assumes there is a linear/straight line relationship between the input features and output predictions. This is an oversimplification for most relationships and immediately, it’s clear that this would cause problems when the relationship between the input and output is non-linear e.g. quadratic. In this case, the results of the model would be insufficient to warrant any benefit, as the it would produce an inaccurate model. Residual plots can be used to check for linearity/non-linearity through searching for patterns in the plot.

Another example of how linear regression may give inaccurate predictions is its sensitivity to outliers. Real life data commonly contains outliers, which is why this can be a problem with a linear model. Outliers can result in the slope of the regression line being changed. Of course, outliers shouldn’t be removed from the data, as this is bad practice. If we take an example dataset which has most of its point in the range (0, 20) on the x axis, with a few outliers at around x = 100, these outliers could have a significant impact on the predictions made around those points, making them inaccurate.

1. Discuss three pros/cons of an SVM classifier vs a neural net classifier.

Both SVMs and neural nets share similar architectures at a low level. Neural networks are a little more complex than SVMs however, as they can drive deeper, with the ability to add more hidden layers. These hidden layers individually work similarly to the kernel function used in an SVM. However, with this ability comes complexity. A pro for neural networks is that they tend to work better when there is an abundance of data, with a complex structure and or high dimensionality. SVM is not suited to large datasets. This stems from the fact that the complexity of the SVM training is influenced heavily by the dataset size. The kernel used in SVMs requires memory dependent on the amount of data in the set. For these reasons SVMs don’t work well with large datasets. SVMs tend to work with smaller subsets of datasets as it can determine decision boundaries accurately.

We know from the points stated above that networks require more training data than SVMs. This is a possible con of neural networks which is that training time is usually much larger for neural networks compared to SVMs. This can be seen as a disadvantage if speed is an issue when solving a classification problem. SVMs tend to have shorter training times mainly as the support vectors for each class contain the hyperplane for the decision, whereas neural networks don’t always do this as decision function is placed at a random location. If the classes in the dataset are well separated, it doesn’t take long to train. As stated before, SVMs work with subsets of the entire dataset. Neural networks however need to analyse the entire dataset in order for it to build an strongly performing network. For these reasons training time is a lot longer for neural networks and can even stretch into days which can make retraining them expensive.

Finally, the optimization methods for parameters used by both SVM and neural networks work differently. Neural networks use gradient descent, which means that sometimes they can be sensitive to the initial hyperparameter weights. In the case where the initial weights place the network too near to the local minimum, which means the network won’t be able to continue learning. SVMs don’t have this problem as they will always converge to a minimum no matter what the initial hyperparameter weights are.

1. Describe the operation of a convolutional layer in a convNet. Give a small

example to illustrate.

Nodes in a convolutional layer use a kernel or filter. They are layers which take a matrix as input and convolve the matrix and the kernel which produces an output matrix. Kernels can come in different sizes and dimensions. The filter is applied to an input matrix or to different feature maps within a deep convolutional network.

**Example:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | 20 | 3-1 | 4 | 5 |
| 51 | 40 | 3-1 | 2 | 1 |
| 11 | 20 | 3-1 | 4 | 5 |
| 5 | 4 | 3 | 2 | 1 |
| 1 | 2 | 3 | 4 | 5 |

**Input matrix** **Kernel**  The kernel is then applied like so:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |
| 5 | 4 | 3 | 2 | 1 |
| 1 | 2 | 3 | 4 | 5 |
| 5 | 4 | 3 | 2 | 1 |
| 1 | 2 | 3 | 4 | 5 |

|  |  |  |
| --- | --- | --- |
| 1 | 0 | -1 |
| 1 | 0 | -1 |
| 1 | 0 | -1 |

Then we iterate through each cell multiplying the kernel value by the input matrix cell value and adding all of these values to get the output:

1x1 + 5x1 + 1x1 + 2x0 + 4x0 +2x0 + 3x-1 + 3x-1 + 3x-1 = -2

The result is added to output matrix:

|  |  |  |
| --- | --- | --- |
| -2 |  |  |
|  |  |  |
|  |  |  |

We then apply the kernel again like so:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 21 | 30 | 4-1 | 5 |
| 5 | 41 | 30 | 2-1 | 1 |
| 1 | 21 | 30 | 4-1 | 5 |
| 5 | 4 | 3 | 2 | 1 |
| 1 | 2 | 3 | 4 | 5 |

Again we calculate the result:

2x1 + 4x1 + 2x1 + 3x0 + 3x0 +3x0 + 4x-1 + 2x-1 + 4x-1 = -2

The result is again added to output matrix:

|  |  |  |
| --- | --- | --- |
| -2 | -2 |  |
|  |  |  |
|  |  |  |

This process is repeated until the output matrix is completed, which would result in the following output matrix:

|  |  |  |
| --- | --- | --- |
| -2 | -2 | -2 |
| 2 | 2 | 2 |
| -2 | -2 | -2 |

Convolutional layers are complete when elements of the output are passed through a non-linearity, after adding a bias normally. They can be combined such that the output of one convolution layer can be the input to another. We can see from the example above that convolutions decrease the size of the input.

1. In k-fold cross-validation a dataset is resampled multiple times. What is the idea behind this resampling i.e. why does resampling allow us to evaluate the generalisation performance of a machine learning model. Give a small example to illustrate.

Resampling is an extremely beneficial method used to help with feature/model selection, parameter optimisation and the overall accuracy of a model. It involves taking small subsets of the larger datasets and performing training and testing. K-fold cross validation helps with evaluating how well models generalise. It begins by splitting the data into k equal sized parts. Then the model is trained using one of these k parts, and tested using the other parts. This is done for each k split which ends up giving us k estimates of J(θ), which can be used to estimate the average and spread of values. It evaluates generalisation performance of our models by using different sets of test data where averages performance scores can be retrieved for more accurate estimates on performance of a model on new data. The following image taken from the slides illustrates the way K-fold splits a dataset into k sets of test and training data as I have explained.

A picture containing bar chart

Description automatically generated

When the test data from the k split is predicted, the accuracy for each split tends to differ. This is usually due to noise within the test and or training data. If the training data is noisy, the learned model parameters change from one set to another. There will be noise changes from one piece of test data to another, if we take for example a coin tossed n/k times, the sequence of heads and tails will be different for each set n/k tosses. Each set of test data will have n/k points, which will produce a prediction accuracy. We can then get the average of these k accuracies which should smooth out the noise, given n/k is large enough(k is small enough).

**Code Appendix:**

import numpy as np

from numpy.core.fromnumeric import size

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.linear\_model import LinearRegression

from sklearn.neighbors import KNeighborsClassifier

from sklearn.dummy import DummyClassifier

from sklearn import linear\_model

from sklearn.model\_selection import KFold

from sklearn.metrics import mean\_squared\_error

from sklearn.linear\_model import LogisticRegression

from sklearn.linear\_model import PoissonRegressor

from sklearn.preprocessing import PolynomialFeatures

def createFeatures():

    dublinBikesData = pd.read\_csv("dublinbikes\_20200101\_20200401.csv")

    avondaleData = dublinBikesData.loc[dublinBikesData['NAME'] == 'AVONDALE ROAD']

    pearseData = dublinBikesData.loc[dublinBikesData['NAME'] == 'PEARSE STREET']

    # Avondale Features

    time = avondaleData.iloc[:,1]

    bikes = avondaleData.iloc[:,6]

    # Pearse Features

    # time = pearseData.iloc[:,1]

    # bikes = pearseData.iloc[:,6]

    convertedTimes = pd.to\_datetime(time)

    datasetSize = size(convertedTimes)

    # plt.scatter(convertedTimes, bikes, s=0.5)

    # plt.legend()

    # plt.show()

    print(type(convertedTimes))

    minute = []; hour = []; day = []; month = []; hourBeforeBikes = []

    previousBike1 = []; previousBike2 = []; previousBike3 = []; noBikes = []; weekBeforeBikes = []; dayBeforeBikes = []; combinedTime = []

    print(convertedTimes.iloc[0])

    for i in range(0,datasetSize):

        noBikes.append(bikes.iloc[i])

        minute.append(convertedTimes.iloc[i].minute)

        hour.append(convertedTimes.iloc[i].hour)

        day.append(convertedTimes.iloc[i].day)

        combinedTime.append(int(str(convertedTimes.iloc[i].hour)+str(convertedTimes.iloc[i].minute)))

        month.append(convertedTimes.iloc[i].month)

        if(i > 2):

            previousBike1.append(bikes.iloc[i-1])

            previousBike2.append(bikes.iloc[i-2])

            previousBike3.append(bikes.iloc[i-3])

        else:

            previousBike1.append(0)

            previousBike2.append(0)

            previousBike3.append(0)

        if(i > 2015):

            weekBeforeBikes.append(bikes.iloc[i-2016])

        else:

            weekBeforeBikes.append(0)

        if(i > 287):

            dayBeforeBikes.append(bikes.iloc[i-288])

        else:

            dayBeforeBikes.append(0)

        if(i > 11):

            hourBeforeBikes.append(bikes.iloc[i-12])

        else:

            hourBeforeBikes.append(0)

    return np.column\_stack((combinedTime, previousBike1, previousBike2, previousBike3, weekBeforeBikes, dayBeforeBikes, hourBeforeBikes)), noBikes

def calculateLassoPerformance(linearOutput, predictions, accuracy\_scores):

    hits = 0; misses = 0

    roundedPredictions = [round(element) for element in predictions]

    for i in range(0,len(roundedPredictions)):

        if(roundedPredictions[i]==linearOutput[i]):

            hits+=1

        else:

            misses+=1

    # print(hits/(hits+misses))

    accuracy = hits/(hits+misses)

    accuracy\_scores.append(accuracy)

def calculateKNNPerformance(linearOutput, predictions, accuracy\_scores):

    hits = 0; misses = 0

    for i in range(0,len(linearOutput)):

        if(predictions[i]==linearOutput[i]):

            hits+=1

        else:

            misses+=1

    # print(hits/(hits+misses))

    accuracy = hits/(hits+misses)

    accuracy\_scores.append(accuracy)

def KLassoReg(features, output):

    Ci\_range = [0.001, 0.01, 0.1, 1, 10, 20]

    mean\_error=[]; std\_error=[]

    arrayOut = np.array(output)

    accuracy\_mean = []; accuracy\_std\_dev = []

    for C in Ci\_range:

        model = linear\_model.Lasso(alpha=1/(2\*C)).fit(features, output)

        temp = []; accuracy\_scores = []

        kf = KFold(n\_splits=5)

        for train, test in kf.split(features):

            model.fit(features[train], arrayOut[train])

            predictions = model.predict(np.array(features[test]))

            calculateLassoPerformance(arrayOut[test], predictions, accuracy\_scores)

            temp.append(mean\_squared\_error(arrayOut[test],predictions))

        mean\_error.append(np.array(temp).mean())

        std\_error.append(np.array(temp).std())

        accuracy\_mean.append(np.array(accuracy\_scores).mean())

        accuracy\_std\_dev.append(np.array(accuracy\_scores).std())

    graphErrorBar(Ci\_range, mean\_error, std\_error, "Lasso - Mean Squared Error", "Mean Squared", "Ci")

    graphErrorBar(Ci\_range, accuracy\_mean, accuracy\_std\_dev, "Lasso - Accuracy Score", "Accuracy", "Ci")

def Kknn(features, output):

    Ki\_range = [1, 2, 3, 4, 5, 6]

    mean\_error=[]; std\_error=[]

    arrayOut = np.array(output)

    accuracy\_mean = []; accuracy\_std\_dev = []

    for n\_neighbours in Ki\_range:

        model = KNeighborsClassifier(n\_neighbours, weights='uniform')

        temp = []; accuracy\_scores = []

        kf = KFold(n\_splits=5)

        for train, test in kf.split(features):

            model.fit(features[train], arrayOut[train])

            predictions = model.predict(np.array(features[test]))

            calculateKNNPerformance(arrayOut[test], predictions, accuracy\_scores)

            temp.append(mean\_squared\_error(arrayOut[test],predictions))

        mean\_error.append(np.array(temp).mean())

        std\_error.append(np.array(temp).std())

        accuracy\_mean.append(np.array(accuracy\_scores).mean())

        accuracy\_std\_dev.append(np.array(accuracy\_scores).std())

    graphErrorBar(Ki\_range, mean\_error, std\_error, "KNN - Mean Squared Error", "Mean Squared", "Ki")

    graphErrorBar(Ki\_range, accuracy\_mean, accuracy\_std\_dev, "KNN - Accuracy Score", "Accuracy", "Ki")

def getModel(features, output, type):

    if(type=="Lasso"):

        model = linear\_model.Lasso(alpha=1/(2\*1), max\_iter=100).fit(features, output)

    elif(type=="Ridge"):

        model = linear\_model.Ridge(alpha=1/(2\*40)).fit(features, output)

    elif(type=="Logistic"):

        model = LogisticRegression(C = 1, penalty="l2", max\_iter=100).fit(features, output)

    elif(type=="Linear"):

        model = LinearRegression().fit(features, output)

    elif(type=="Poisson"):

        model = PoissonRegressor(max\_iter=100, alpha=0.5).fit(features,output)

    elif(type=="KNN"):

        model = KNeighborsClassifier(5, weights='uniform').fit(features, output)

    elif(type=="Baseline"):

        model = DummyClassifier(strategy="uniform").fit(features, output)

    return model

def qStepModel(step\_size, features, output):

    rowIndex = 2016; hits = 0; misses = 0; difference = 0

    while rowIndex < 20300:

        x\_train = features[rowIndex-2016:rowIndex]

        y\_train = output[rowIndex-2016:rowIndex]

        x\_test = features[rowIndex+1:rowIndex+step\_size+1]

        expected\_result = output[rowIndex+1:rowIndex+step\_size+1]

        # poly = PolynomialFeatures(3)

        # xTrainPoly = poly.fit\_transform(x\_train)

        model = getModel(x\_train, y\_train, "Lasso")

        # xTestPoly = poly.fit\_transform(x\_test)

        print("Row: ", rowIndex)

        predictions = [0, 0, 0]

        index = 0

        for element in x\_test:

            if(len(predictions)>3):

                predictions.pop(0)

            if((index+1)==len(x\_test)):

                if(index==1):

                    element[4] = predictions[2]

                else:

                    if(predictions[0]<100000 and predictions[0]>-10):

                        element[1] = predictions[0]

                    if(predictions[1]<100000 and predictions[1]>-10):

                        element[2] = predictions[1]

                    if(predictions[2]<100000 and predictions[2]>-10):

                        element[3] = predictions[2]

                t = model.predict([element])

                targetPrediction = 0

                if(t[0]<100000  and t[0]>-10):

                    targetPrediction = round(t[0])

                if(targetPrediction==expected\_result[step\_size-1]):

                    hits+=1

                else:

                    diff = abs(targetPrediction-expected\_result[step\_size-1])

                    if(diff>-10 and diff<1000000):

                        difference+=diff

                    misses+=1

            else:

                if(index!=0):

                    if(predictions[2]<100000 and predictions[2]>-10):

                        if(index==1):

                            element[3] = predictions[2]

                        elif(index==2):

                            element[2] = predictions[1]

                            element[3] = predictions[2]

                        else:

                            element[1] = predictions[0]

                            element[2] = predictions[1]

                            element[3] = predictions[2]

                        t = model.predict([element])

                        feedbackPrediction = round(t[0])

                        predictions.append(feedbackPrediction)

                else:

                    t = model.predict([element])

                    feedbackPrediction = 0

                    if(t[0]<100000 and t[0]>-10):

                        feedbackPrediction = round(t[0])

                    predictions.append(feedbackPrediction)

            index+=1

        rowIndex += 1

    print("Hits: ", hits)

    print("Misses: ", misses)

    print("Average Hits: ", hits/(hits+misses))

    print("Average Miss Size: ", difference/(hits+misses))

def graphErrorBar(Ci\_range, mean\_error, std\_error, title, label, xLabel):

    plt.title(title)

    plt.plot(Ci\_range, mean\_error)

    plt.errorbar(Ci\_range, mean\_error, yerr=std\_error, fmt ='ro', label=label)

    plt.xlabel(xLabel)

    plt.ylabel("Accuracy")

    plt.legend()

    plt.show()

features, output = createFeatures()

# 5 mins every row

# 10 mins = 2 rows

# 30 mins = 6 rows

# 60 mins = 12 rows

qStepModel(2, features, output)

# qStepModel(6, features, output)

# qStepModel(12, features, output)

# K Fold Cross Validation

# KLassoReg(features, output)

# Kknn(features, output)