Brendan Jobus
18321740

1. Predicting bike station occupancy using the methods from this module is not entirely difficult, however, also not especially accurate. The methodology that we learnt through this module gives us the capability to design a *feed-forward* model, or a model that simply uses the data that we currently have to predict whats going to happen next. This isn't especially complicated to implement, however, by simply predicting whats going to happen later on, we might overlook some other possibilities that could occur in the mean time; this isn't a huge problem for us when we are simply predicting 5 or 10 minutes into the future, however, as we start to predict further and further ahead, our model will become less and less accurate. While our *feed-forward* model might not be great at predicting or taking into account these things, a *feed-back* model, which uses smaller predictions as inputs into our model for our next preiction, will exacerbate the mistakes that we make after each prediction, making them far more difficult to design.

For my models, I decided that I would use a basic Ridge regression model and a more complicated Random Forest regressor model, the advantages of the Ridge regression model is that it is far simpler to create, is much easier to gain an insight into how it works, due to it being possible to look into the weights assigned to each parameter, and it being very quick to train, the downsides, however, are that it isn't very accurate in comparison to more complex models when dealing with very complicated systems. Random Forest Regressor is a machine learning algorithm that looks to improve the decision tree regressor model, it does this by making a large number of trees (hence the name Forest) and training each of them with slightly different datasets. The datasets are created by bootstraping, i.e. taking random samples with replacement out of the original dataset, and usually takes as many samples as the size of the dataset. By learning from slightly differing datasets, it tries to decrease overfitting and the effect of outliers on the model. The predictions are made by running each decision tree and then averaging the all of the answers. In sklearn's RandomForestRegressor model, n_estimators, or simply estimators as I call them in this report, are the number of trees the model uses.

I used R squared scores, as well as Root Mean Squared Error(RMSE) to decide upon what hyperparameters and features to use, and also used Meas Absolute Error(MAE) when comparing the finished models. I also used cross validation whenever a performance score was calculated, however, you can't use Kfold cross validation with time series models, this is due to the fact that training a model with data that comes after data that we would be predicting is non-sensical from a logical point of view, and can bias our results. So instead, I used the sklearn TimeSeriesSplit module to implement cross validation. This module implements cross validation by spliting the data as usual, but instead of using all the data except for one section to train the model, and then the section that wasn't used to test the model, it instead, in the case where we use k = 5, uses the first section to train the model and the second for testing the model, and then in the next step, we use the first two sections for training and the third for testing, and continue in that way, where the next training set is the last training set combined with the last test set, and the new test set is just the next section. This allows us to use regular cross validation, that stops us from accidentally training on biased data, on a time series model.

The first thing I did after I got my data, was to look through it to see what stations I was going to select, and to see what I was going to have to clean up in the data. To select the stations, due to my lack of knowledge of Dublin city, I decided that I would select a station in the city centre, and then use the longitude and latitude as coordinates and implement a eculidean distance function which would give me the distances of all the stations from the city centre station I chose, and then simply take the station furthest from my original station, using this method, I decided upon Blessington Street and Kilmainham Gaol as the two stations I was would use in this assignment. From there, I worked on cleaning the data, there were two main issues with the data that we got, the first was that there were large gaps in the data, some times there would be days or even weeks of data missing, to solve this, I decided that it would be best to simply use the implementation from the notes and go from February 4th to March 14th. Then, I had to transform the dates from strings to unix integer format, and finally, I had to extract our targets, which is the current occupancy of the station, which is simply the maximum number of bike stands in the station minus the available bike stands.

When designing my Ridge regression model, I had to decide upon whether I wanted it to take in the data for both dates at once or not, if the model were to take in dates, I would simply have to use one hot encoding to turn the name of the stations into essentially two booleans, each with different weights that would effect the output, however, as these two stations were very far away and not guarenteed to have very similar trends, I decided that they should be seperated instead, and when looking at the difference between the weights of two basic models trained with each set to predict 10 minutes ahead, we see that the models weigth even the basic dataset very differently.

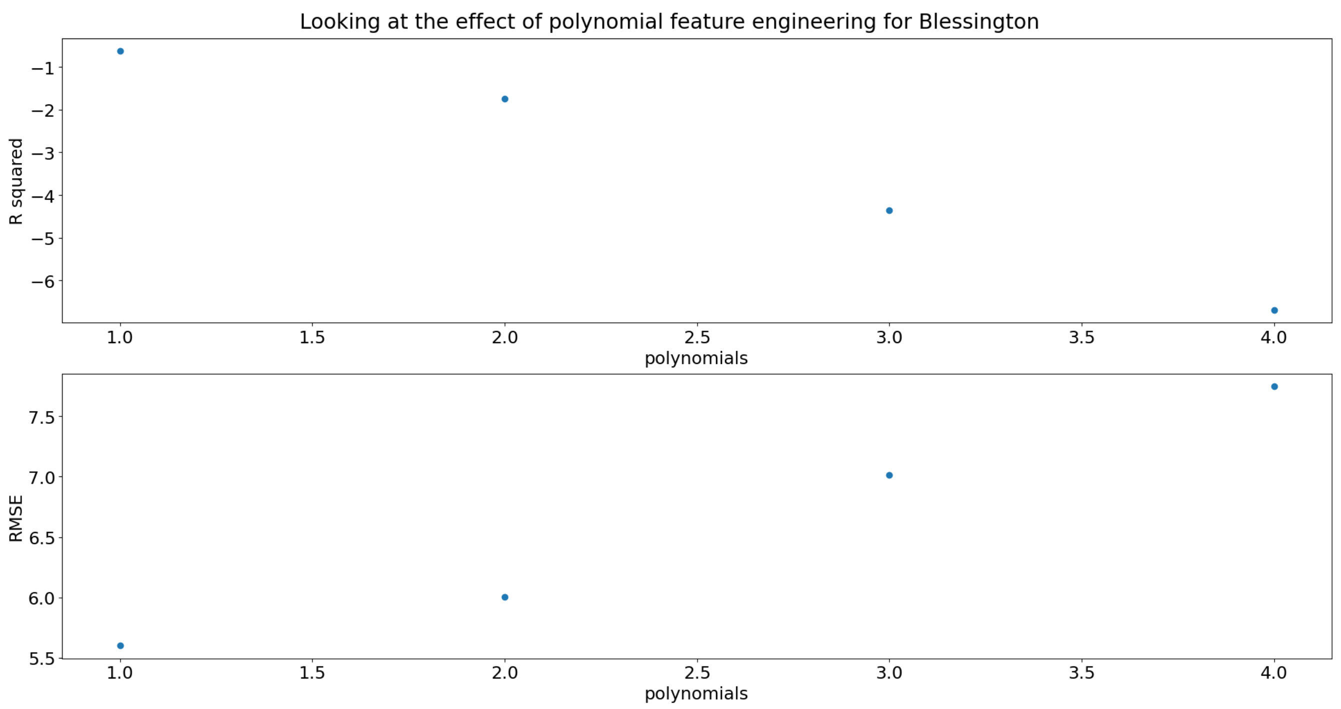
Station	Index	Station ID	Time	Last Update	Bike Stands	Available Stands	Available Bikes
Blessington	-5.8874	0	87.204245	79.7463	0	-2.2529	1.02354449
Kilmainham	-20.8597	0	4.04445556	3.914544	0	-8.12205	-42.016

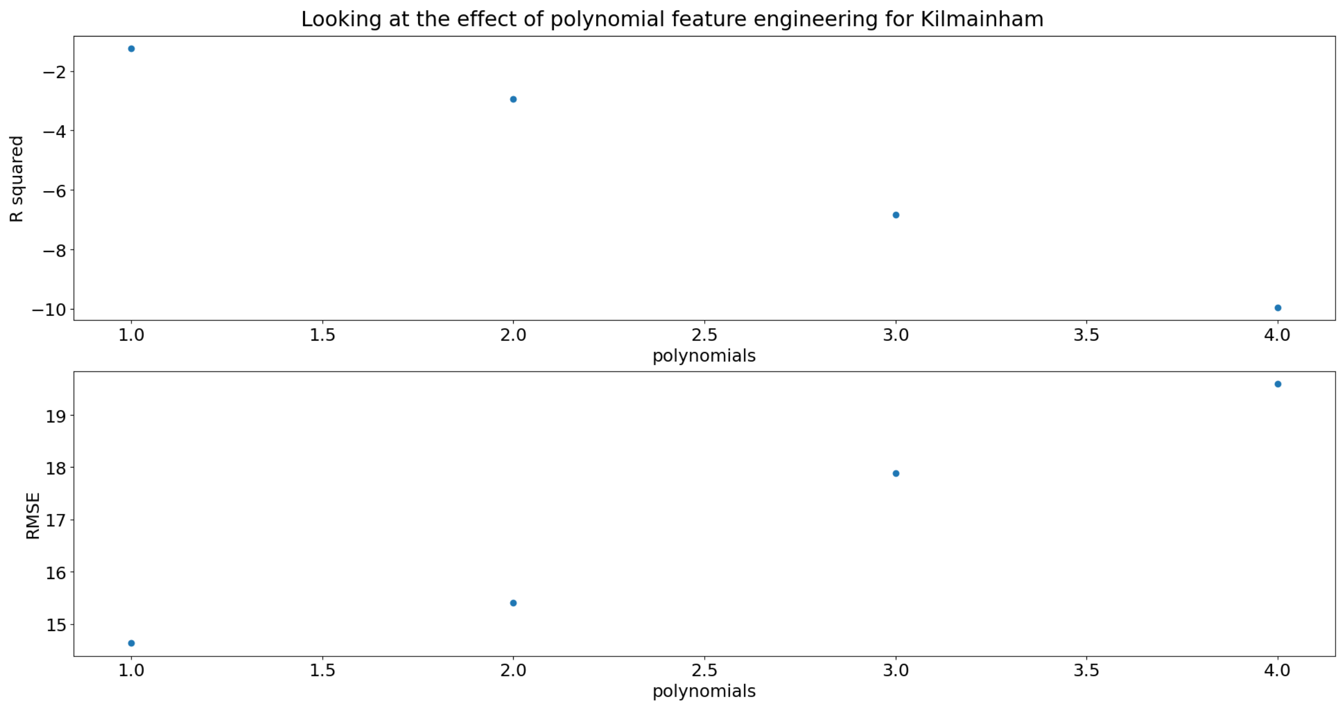
Station	Status	Latitude	Longitude	Time In Days	Data from 3 updates ago	Data from 2 updates ago	Data from 1 update ago
Blessington	0	0	$1.21x10^{-11}$	0.872042	-0.003245	-0.8911367	2.2529
Kilmainham	0	$-2.55x10^{-10}$	$-3.19x10^{-11}$	4.0444	-2.8125	-1.428	8.122

And after removing the data that gave either no input or as close as possible to no input, as well as the index column, which existed from the original csv file, and shouldn't give any input into the data inself, I got weights like this:

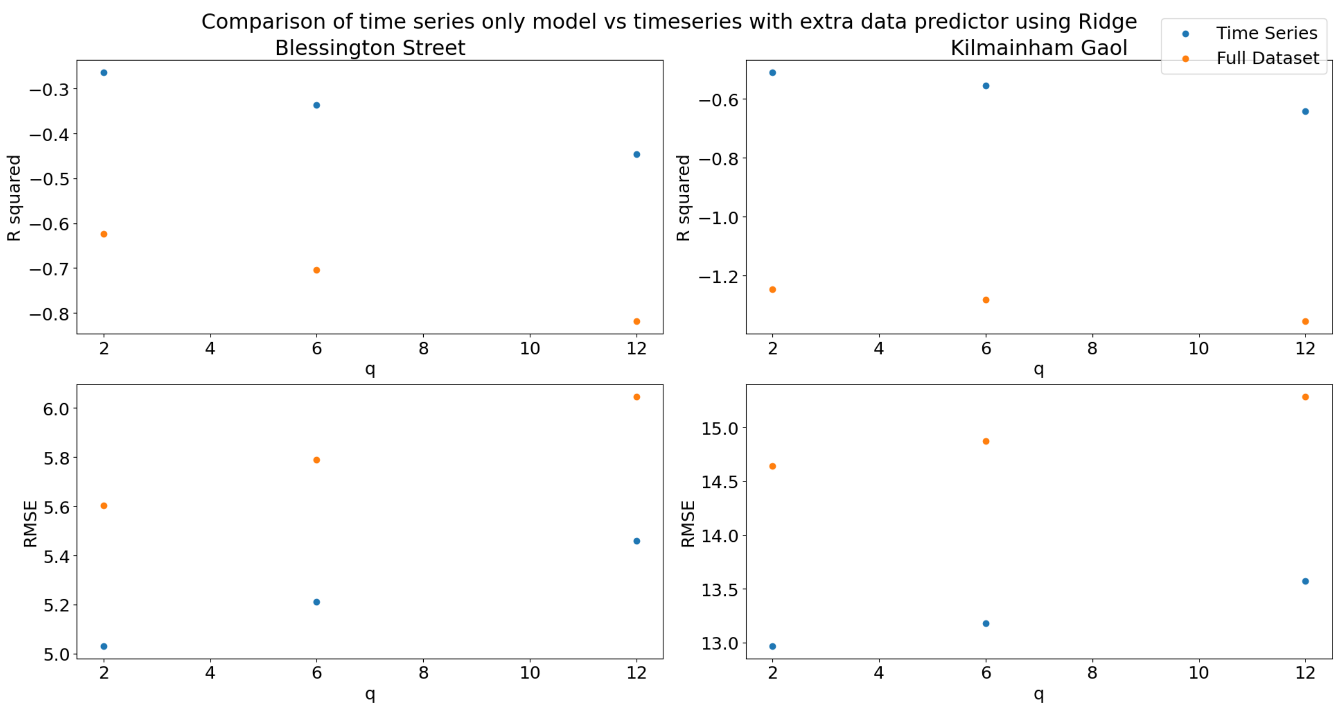
Station	Time	Last Update	Available Stands	Available Bikes	Time In Days	Data from 3 updates ago	Data from 2 updates ago	Data from 1 update ago
Blessington	-1.087	-1.1609	-2.2676	1.0201	-1.08781	-0.0663	-0.88473	2.2676
Kilmainham	-2.910	-2.9987	-9.8117	-3.2158	-2.91055	-3.5736	-1.374552	9.8117

From here, I wanted to see if using polynomial feature engineering would help our model, so I used the polynomial features sklearn module to test different polynomial feature engineering.

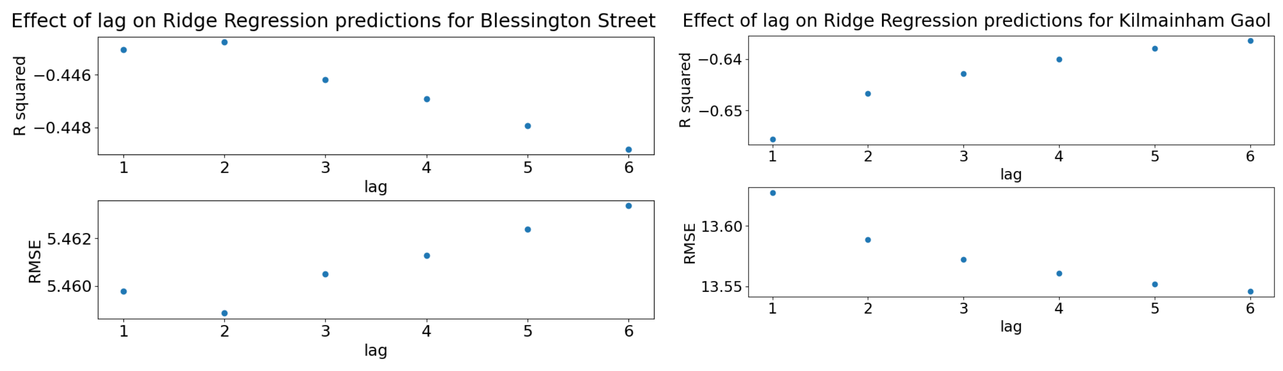


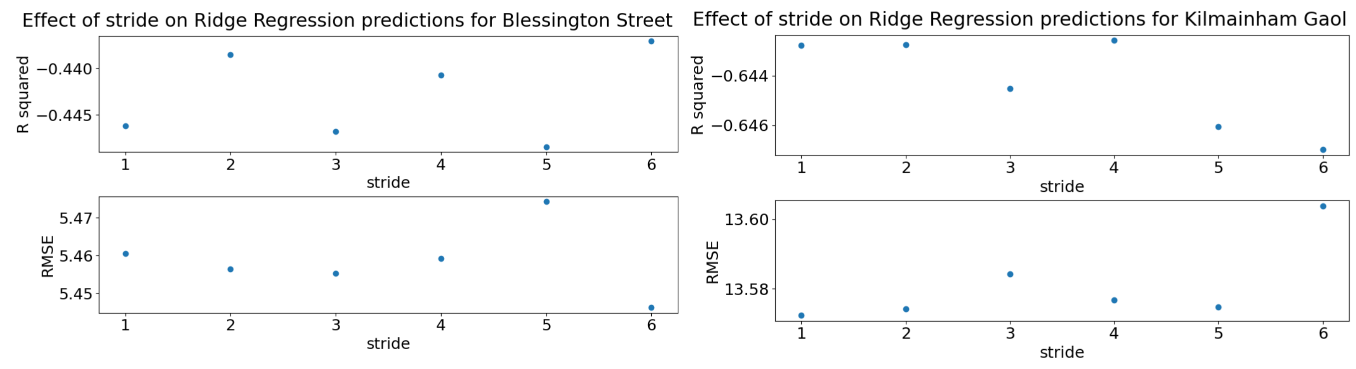


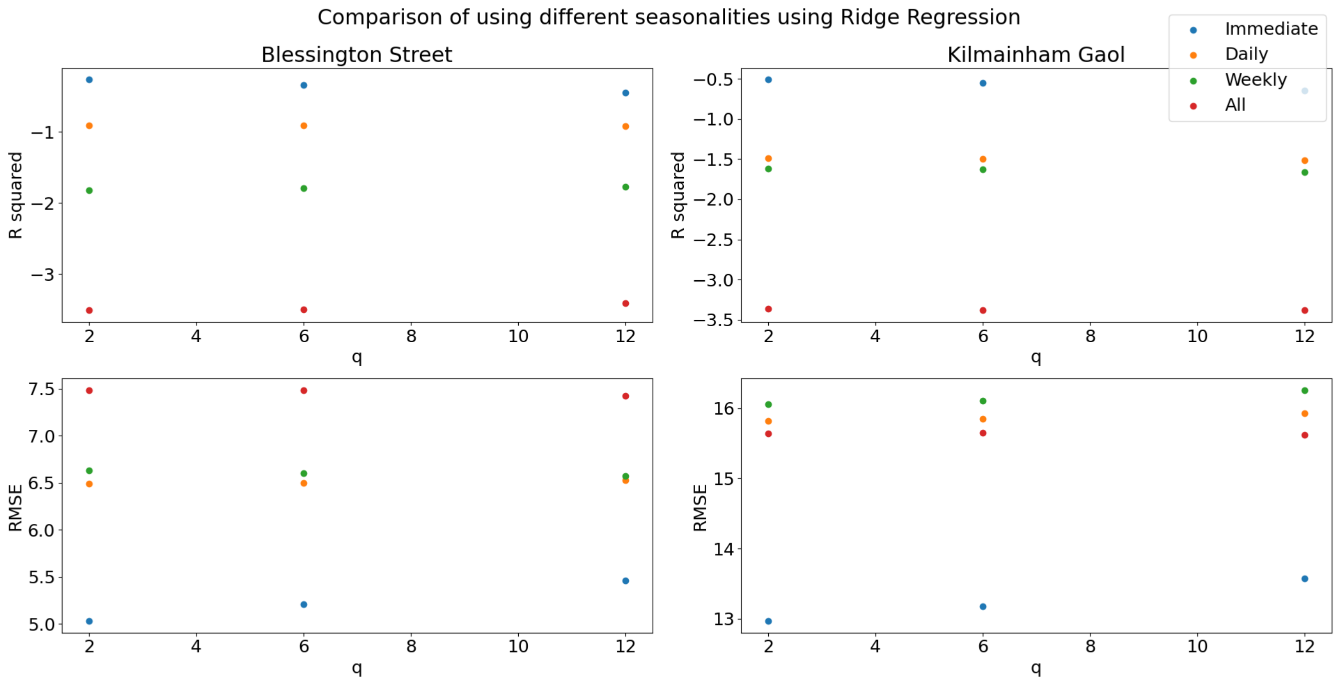
This told me very clearly that the polynomial engineering would not be helpful here. I then decided to test using the features that I currently have, against just using the time series data I was generating to see if the features were actually helping with our predictions.



With this we can see that for both stations and for all q steps ahead, using just time series data extracted from the target is far more effective for predicting the occupancy. After this I went to selecting hyper parameters, there are three hyper parameters for a Ridge regression model in a time series scenario; lag, stride and seasonality. The lag is the amount of past data we are going to use, so with a lag of 3, we would use the last 3 targets. The stride is whether, when selecting the past targets to use, we use the immediate last *x* targets, or if we use *x* past targets, with a gap of 3 between each for example. Finally, the seasonality is using past targets from times that might hold some intrinsic information about the current prediction we are trying to make.





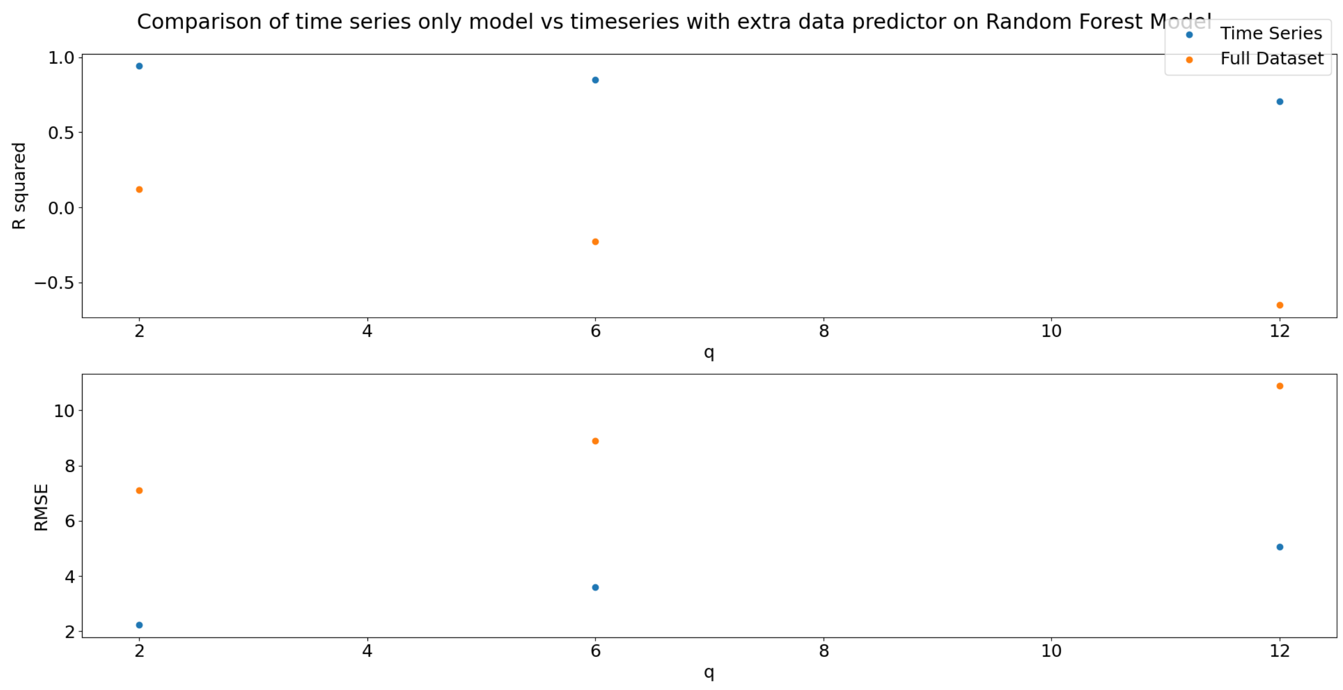


Using this data, I decided that for the Blessington Street model, I would use a lag of 2 and a lag of 6 for the Kilmainham model, a stride of 6 for Blessington and 1 for Kilmainham and for both I would use an immediate seasonality, or simply, just the data in the immediate past.

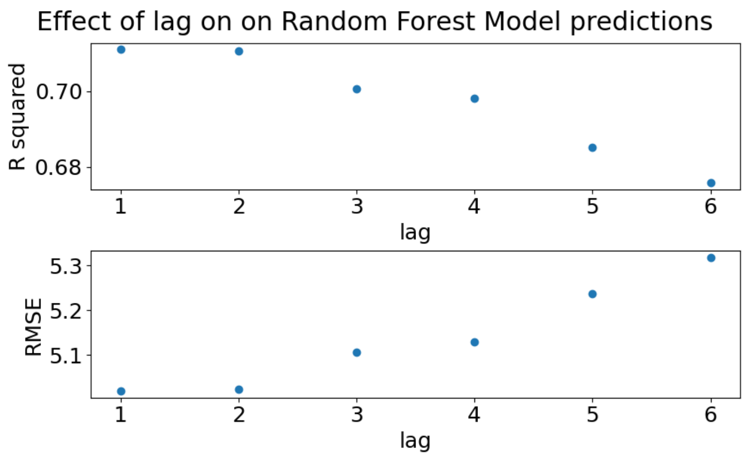
Random Forest Model:

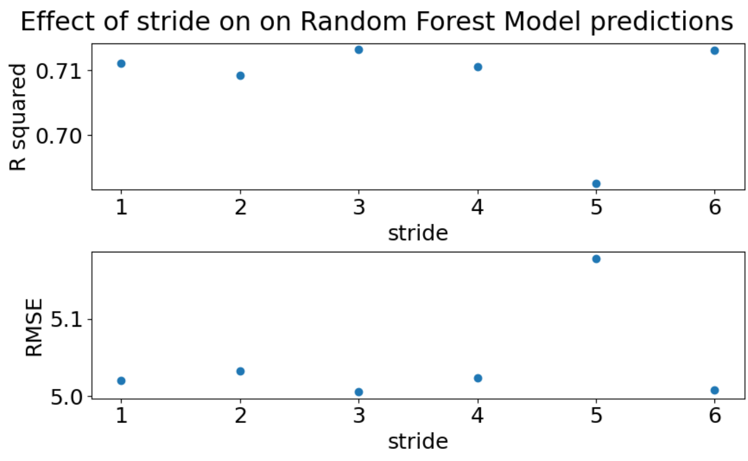
A quick note, I did not seperate the two stations for this model, and the results were r squared scores close to 1 for predicting 10 minutes ahead, and for an hour ahead it just dipped below 0.8, at the time that I was designing the model, I thought this made sense so long as I kept the station id data, but looking back now, while I have no more time to correct the issues, I see that there is a large issue with this, namely that when getting past data and using that as an input to inform our decisions, sometimes, this data will come from the other station. This really shows how, even though random forest models are often significantly more accurate than standard regression models, they are black boxs that tell us very little about what is actually happening to get out output, and thus leave us with the possibility that something wonky could happen and we could get some weird outputs.

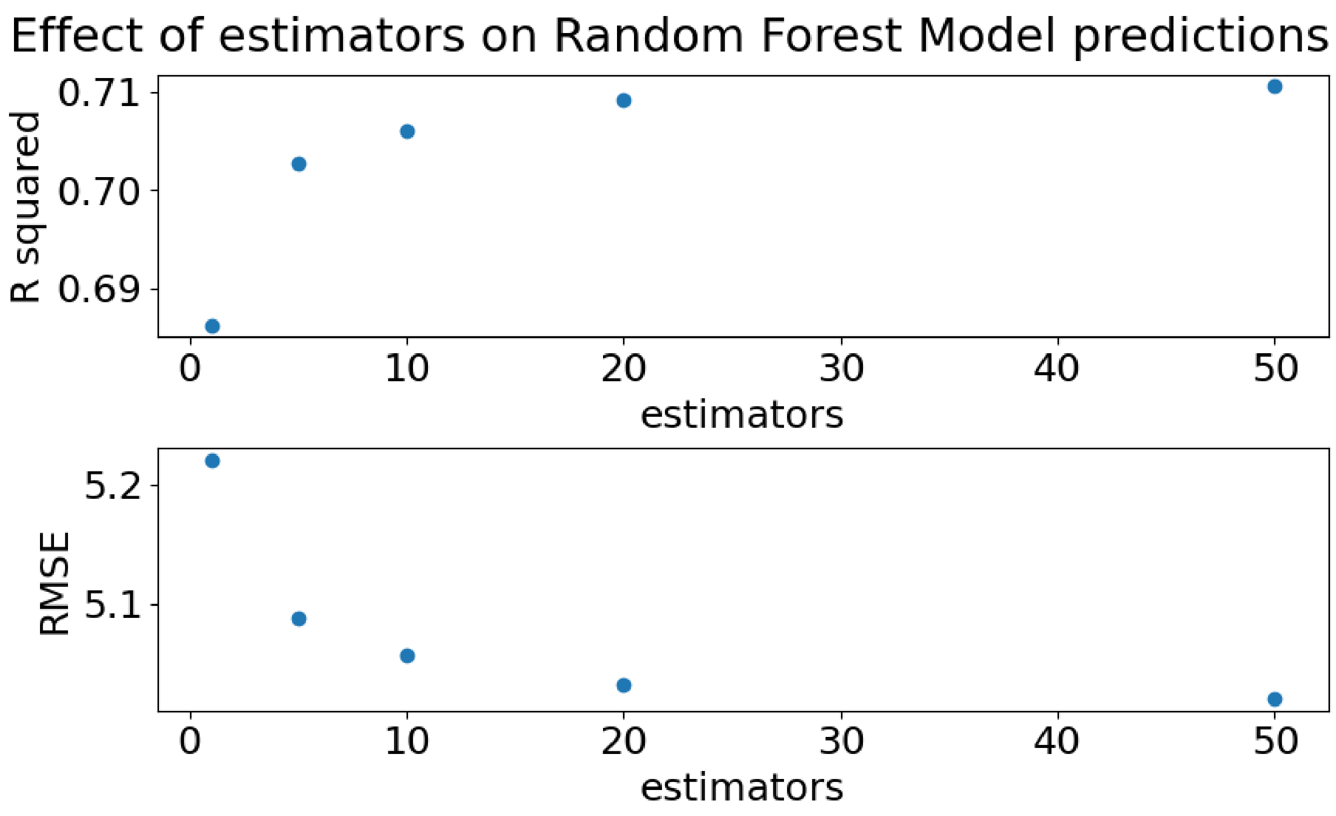
First I looked at whether I should use lots of data from the dataset we downloaded, or just the station id data, so tested them and found that for all the predictions that we want, the extra data actually made our model significantly less accurate, I did do some further testing to see if any other combinations added to the model, but none did and I don't have any room for more graphs.



From there I looked at the hyper parameters, this time, we have estimators, on top of lag and stride.

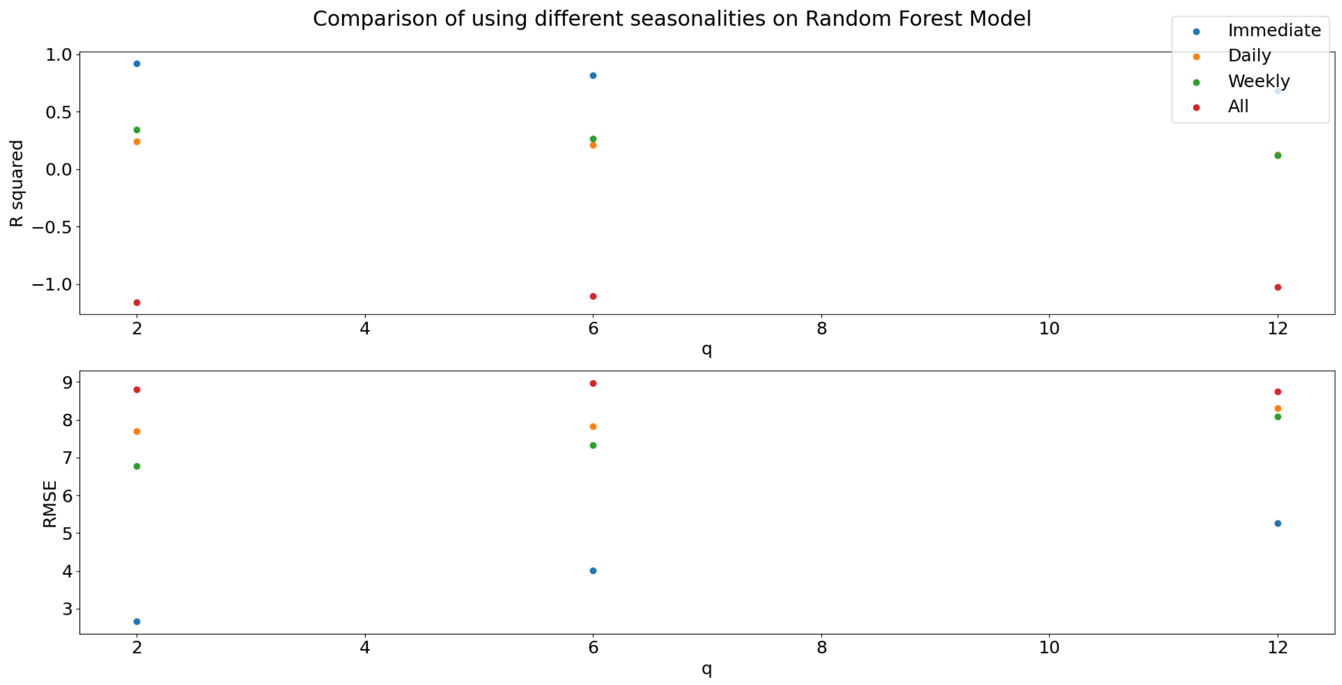




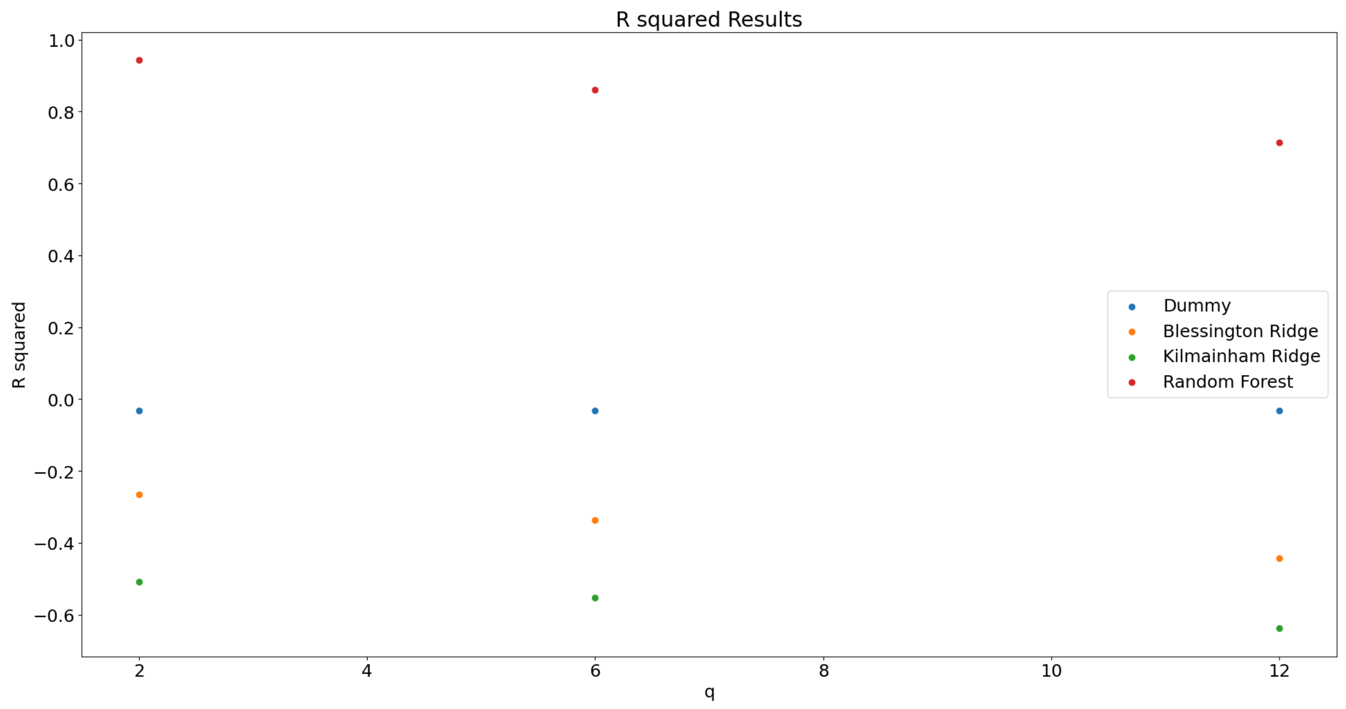


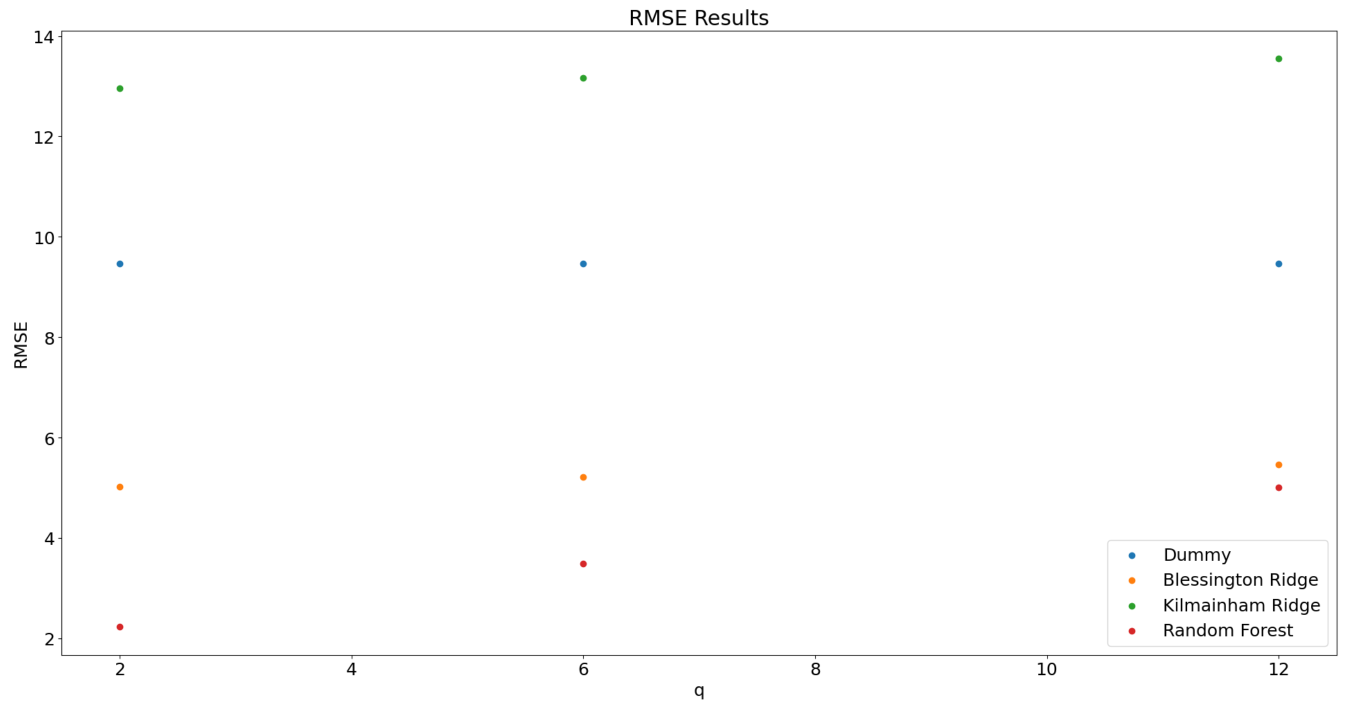
For lag, I decided upon a value of 1 as being the best hyperparameter, as it will reduce the complexity of the model with roughly the same performance as a value of 2. For the stride, I went for a value of 3, and for the estimators, there was not a significant enough improvement going from 20 to 50 estimators for me to to choose 50, as it makes the model significantly more computationaly expensive.

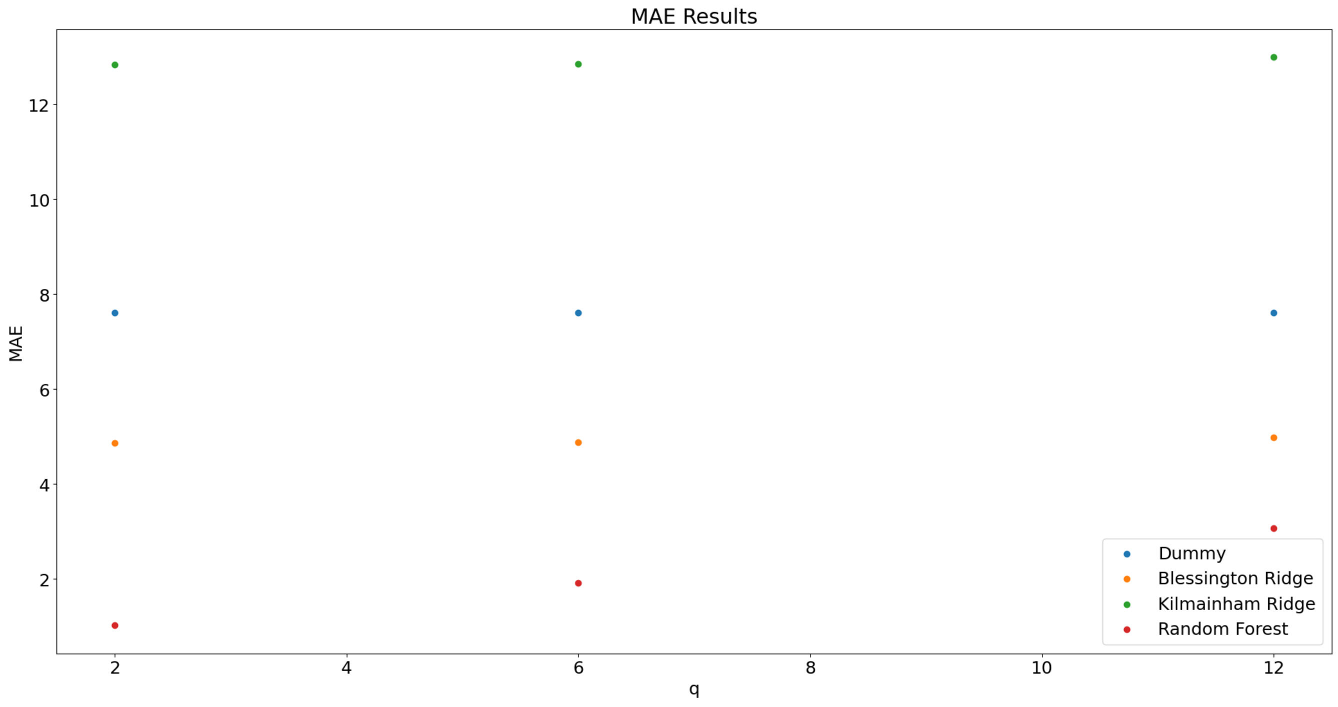
Finally, I looked at the seasonalities again, and once again found that an immediate seasonality was the best option.



Now, I have my three models, to test them, I ran all of them, with the hyper parameters that I found to be the best for each one, against a dummy model that always picks the mean occupancy. The results I got were interesting, both of the ridge models were worse at predicting the occupancy of the bike stands than the dummy model at any point, making them completely useless, however, the random forest model, performed very well and was by far the best model.







2. (i) An ROC curve is a graph that plots the True positive rate and False positive rate of a classifier model, where the curve is obtained by varying some value β, we can use this to evaluate the performance of a classifier, by seeing if any value of β is close to the top left corner of the graph, and at the very least, if it is above a line drawn from the bottom left to top right corner of the graph, this line effectively represents picking the mean everytime. A model which has a value in the top left has very high true positives and very low false positives, making it a good model.

(ii) The first situation is when the differential of our model is past quintic, i.e. is a curve with multiple local minimums, we can get innacurate results here if the minimisation that we arrive at is not the global minimum but rather a local minimum, this would make our model settle on weights that are not actually the weights that minimise the cost function and thus give relatively poor and inaccurate results. The second would be if the data taken in was both not normalised and had outliers, this would cause the weights for the variable that the outliers belong to to penalise normal values too much, or it would allow he outlier to affect the data too much.

(iii) Firstly, an SVM is a much more open system that allows us to have more insight into what is being done to the inputs to obtain the output, we can look at each individual weight and see what is being done, however, with a neural network, we can't see inside of the hidden layers and so, don't quite know what is being done to obtain the results.

A disadvantage for an SVM, however, is the time it takes to use the model after training it, this is because, in comparison to a nerual network, with uses known matrix operations on each layer, making each layer highly parrelizable, an SVM uses basic addition/subtraction and multiplication and is usually a serial process, meaning that as we try to solve larger and larger problems, an SVM becomes much more computationally expensive, whereas the parrelizable neural net is relatively light.

Also, because neural networks use gradient descent and SVMs don't, neural nets are vulnerable to finding a minimization that is local and not a global minimum, where as SVMs are always guarenteed to converge on a global minimum.

(iv) The convolutional layer of a convNet uses a matrix called a kernel which is used to mulitply against values in the input matrix, and then reduce the output matrix to a single value, if the size of the kernel is the same as the input, we get a single value, however, if the size of the kernel is smaller than the kernel, then we have to do this operation multiple times, until we have done this to all the values in the input, giving us another matrix as the output. The way in which we apply this process to an input matrix that is larger than the kernel, is essentially a window method, we start at the top left element of the matrix, and apply the kernel to the input, once we've done this, we move across the rows till we have applied the kernel to all the elements in the row, once we've done this we go back to the left most elements, but this time go down one row and repeat the above process, we do this until every element in the input matrix has been involved in a convolution.

$M = \begin{vmatrix} 1 & 3 & 5 \\ 2 & 7 & 8 \\ 6 & 9 & 3 \end{vmatrix} \qquad K = \begin{vmatrix} 2 & 1 \\ 3 & 1 \end{vmatrix} \qquad O = \begin{vmatrix} & \\ & \end{vmatrix}$

First step:
(2x1) + (1x3) + (3x2) + (1x7) = 18

$O = \begin{vmatrix} 18 & \\ & \end{vmatrix}$

Second step:
(2x3) + (1x5) + (3x7) + (1x8) = 40

$O = \begin{vmatrix} 18 & 40 \\ & \end{vmatrix}$

And so on till you get:

$O = \begin{vmatrix} 18 & 40 \\ 38 & 52 \end{vmatrix}$

(v) By resampling multiple times, we attempt to remove the possibility of the data that we are training on has some trend that is not representative of reality, for example, if we had a set [1, 2, 4, 5, 6, 5, 4, 3, 2, 1] and we tested our model with 20% of the set, we could get a training set of [4, 5, 6, 5, 4, 3, 2, 1], which would train our model with data that would lean more towards a desending set of numbers, rather than the reality which is a set of numbers that follow a quadratic trend.

Appendix:

```
import math
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.dummy import DummyRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.model_selection import TimeSeriesSplit, train_test_split
from sklearn.preprocessing import PolynomialFeatures, StandardScaler

plt.rc('font', size=18); plt.rcParams['figure.constrained_layout.use'] = True

def euclidAbsDistance(baseLat, baseLong, xLat, xLong):
return abs(math.sqrt((baseLat - xLat) ** 2 + (baseLong - xLong) ** 2))

def reduceStations():
data = pd.read_csv('./data/dublinbikes_20200101_20200401.csv')

allGeocodes = data.copy()
allGeocodes = allGeocodes.drop_duplicates(subset=['STATION ID'])
allGeocodes = allGeocodes[ ['STATION ID', 'LATITUDE', 'LONGITUDE'] ]
base = allGeocodes.loc[allGeocodes['STATION ID'] == 2]
baseLat = base['LATITUDE'][0]; baseLong = base['LONGITUDE'][0]
allGeocodes = allGeocodes.drop(index=0)
dists = allGeocodes.apply(lambda x : euclidAbsDistance(baseLat, baseLong, x['LATITUDE'],
x['LONGITUDE']), axis=1)
furthestStation = data.iloc[dists.idxmax()]['STATION ID']
reducedData = data.loc[(data['STATION ID'] == 2) | (data['STATION ID'] ==
furthestStation)].copy()

## before we write we need to add in a column for bikes in station
maxBikes = reducedData['BIKE STANDS']
availableSpots = reducedData['AVAILABLE BIKE STANDS']
currentOccupancy = maxBikes - availableSpots
reducedData['CURRENT OCCUPANCY'] = currentOccupancy
reducedData.to_csv('./data/bike_data.csv')

def extractOccupancyOnly(data, station):
df = data.loc[(data['NAME'] == station)].copy()
start = pd.to_datetime("04-02-2020", format='%d-%m-%Y')
end = pd.to_datetime("14-03-2020", format='%d-%m-%Y')
t_full = pd.array(pd.DatetimeIndex(df.loc[:, 'TIME']).view(np.int64)) / 1000000000
dt = t_full[1] - t_full[0]
t_start = pd.DatetimeIndex([start]).view(np.int64) / 1000000000
t_end = pd.DatetimeIndex([end]).view(np.int64) / 1000000000
```

```
y = np.extract([(t_full >= t_start) & (t_full <= t_end)], df.loc[:, 'CURRENT
OCCUPANCY']).view(np.int64)

df = data.copy()
start = pd.to_datetime("04-02-2020", format='%d-%m-%Y')
end = pd.to_datetime("14-03-2020", format='%d-%m-%Y')
t_full = pd.array(pd.DatetimeIndex(df.loc[:, 'TIME']).view(np.int64)) / 1000000000
dt = t_full[1] - t_full[0]
t_start = pd.DatetimeIndex([start]).view(np.int64) / 1000000000
t_end = pd.DatetimeIndex([end]).view(np.int64) / 1000000000
fullY = np.extract([(t_full >= t_start) & (t_full <= t_end)], df.loc[:, 'CURRENT
OCCUPANCY']).view(np.int64)

return y, fullY, dt

def transformData(data, station):
df = data.loc[(data['NAME'] == station)].copy()
# first transform all the data, and then take just the data that is in t
t = pd.array(pd.DatetimeIndex(df.loc[:, 'TIME']).view(np.int64)) / 1000000000
df['TIME'] = t;

start = pd.to_datetime("04-02-2020", format='%d-%m-%Y')
end = pd.to_datetime("14-03-2020", format='%d-%m-%Y')
t_full = t
dt = t_full[1] - t_full[0]
t_start = pd.DatetimeIndex([start]).view(np.int64) / 1000000000
t_end = pd.DatetimeIndex([end]).view(np.int64) / 1000000000
t = np.extract([(t_full >= t_start) & (t_full <= t_end)], t_full)
y = np.extract([(t_full >= t_start) & (t_full <= t_end)], df.loc[:, 'CURRENT
OCCUPANCY']).view(np.int64)
print('data sampling interval is %d secs' %dt)

df = df[df['TIME'].isin(t)]
lastUpdated = pd.array(pd.DatetimeIndex(df.loc[:, 'LAST UPDATED']).view(np.int64)) /
1000000000
df['LAST UPDATED'] = lastUpdated
df = df.sort_values(by=['TIME'])
t = (t - t[0]) / 60 / 60 / 24

# Turn STATUS into binary: 1 = Open, 0 = Closed
status = pd.array(df['STATUS'])
status = [1 if x == 'Open' else 0 for x in status]
df['STATUS'] = status

# need to add t back onto the features
df = df.drop(['CURRENT OCCUPANCY'], axis=1); df = df.drop(['NAME'], axis=1); df =
df.drop(['ADDRESS'], axis=1)
```

```
df['TIME IN DAYS'] = t

d = data.copy()
t = pd.array(pd.DatetimeIndex(d.loc[:, 'TIME']).view(np.int64)) / 1000000000
d['TIME'] = t;

start = pd.to_datetime("04-02-2020", format='%d-%m-%Y')
end = pd.to_datetime("14-03-2020", format='%d-%m-%Y')
t_full = t
t_start = pd.DatetimeIndex([start]).view(np.int64) / 1000000000
t_end = pd.DatetimeIndex([end]).view(np.int64) / 1000000000
t = np.extract([(t_full >= t_start) & (t_full <= t_end)], t_full)
y = np.extract([(t_full >= t_start) & (t_full <= t_end)], d.loc[:, 'CURRENT
OCCUPANCY']).view(np.int64)

d = d[d['TIME'].isin(t)]
lastUpdated = pd.array(pd.DatetimeIndex(d.loc[:, 'LAST UPDATED']).view(np.int64)) /
1000000000
d['LAST UPDATED'] = lastUpdated
d = d.sort_values(by=['TIME'])
t = (t - t[0]) / 60 / 60 / 24

# Turn STATUS into binary: 1 = Open, 0 = Closed
status = pd.array(d['STATUS'])
status = [1 if x == 'Open' else 0 for x in status]
d['STATUS'] = status

# need to add t back onto the features
d = d.drop(['CURRENT OCCUPANCY'], axis=1); d = d.drop(['NAME'], axis=1); d =
d.drop(['ADDRESS'], axis=1)
d['TIME IN DAYS'] = t

return df, d

def plotResults(x, r2s, rmses, xlabel, title):
fig, (ax1, ax2) = plt.subplots(2)
fig.suptitle(title)
ax1.scatter(x=x, y=r2s); ax1.set(xlabel=xlabel, ylabel='R squared')
ax2.scatter(x=x, y=rmses); ax2.set(xlabel=xlabel, ylabel='RMSE')
fig.show()

def predictor(y, q, dd, lag, modelType='Ridge', x=0, stride=1, estimators=20, prnt=True,
evl=False):
XX = 0

scaledX=0
if type(x) == type(pd.DataFrame()):
```

```
XX = y[0 : y.size - q - lag * dd : stride]
for i in range(1, lag):
X = y[i * dd : y.size - q - (lag - i) * dd : stride]
XX = np.vstack((XX, X))
x = x.iloc[(lag - 1) * dd + q : y.size - (1 * dd) : stride].copy()
if lag == 1:
x['BIKE HISTORY'] = XX
else:
for i, X in enumerate(XX):
x['BIKE HISTORY ' + str(i + 1)] = X
scaled_features = StandardScaler().fit_transform(x.values)
scaledX = pd.DataFrame(scaled_features, columns=x.columns)
else:
XX = y[0 : y.size - q - lag * dd : stride]
for i in range(1, lag):
X = y[i * dd : y.size - q - (lag - i) * dd : stride]
XX = np.column_stack((XX, X))

yy = y[lag * dd + q :: stride]

tscv = TimeSeriesSplit()
if lag == 1:
featureTotals = 0
else:
featureTotals = [0] * XX.shape[1]

if lag == 1:
XX = StandardScaler().fit_transform(XX.reshape(-1, 1))
else:
XX = StandardScaler().fit_transform(XX)

r2Total = 0
rmseTotal = 0
maeTotal = 0
if modelType == 'Ridge':
for train, test in tscv.split(XX, yy):
model = Ridge(fit_intercept=False).fit(XX[train], yy[train])
featureTotals += model.coef_
y_pred = model.predict(XX[test])
r2Total += r2_score(yy[test], y_pred); rmseTotal += np.sqrt(mean_squared_error(yy[test],
y_pred)); maeTotal += mean_absolute_error(yy[test], y_pred)
if prnt == True:
print(featureTotals / ([5] * XX.shape[1]))
else:
for train, test in tscv.split(scaledX, yy):
model = RandomForestRegressor(n_estimators=estimators,
random_state=42).fit(scaledX.iloc[train], yy[train])
```

```
y_pred = model.predict(scaledX.iloc[test])
r2Total += r2_score(yy[test], y_pred); rmseTotal += np.sqrt(mean_squared_error(yy[test],
y_pred)); maeTotal += mean_absolute_error(yy[test], y_pred)

if evl:
return r2Total / 5, rmseTotal / 5, maeTotal / 5
return r2Total / 5, rmseTotal / 5

def fullPredictor(x, y, q, dd, lag, modelType='Ridge', stride=1, estimators=20):
x = x.iloc[(lag - 1) * dd + q : y.size - (1 * dd) : stride].copy()

XX = y[0 : y.size - q - lag * dd : stride]
for i in range(1, lag):
X = y[i * dd : y.size - q - (lag - i) * dd : stride]
XX = np.vstack((XX, X))

if lag == 1:
x['BIKE HISTORY'] = XX
else:
for i, X in enumerate(XX):
x['BIKE HISTORY ' + str(i + 1)] = X
scaled_features = StandardScaler().fit_transform(x.values)
scaledX = pd.DataFrame(scaled_features, columns=x.columns)
yy = y[lag * dd + q :: stride]

tscv = TimeSeriesSplit()
featureTotals = [0] * scaledX.shape[1]
r2Total = 0
rmseTotal = 0
if modelType == 'Ridge':
for train, test in tscv.split(scaledX, yy):
model = Ridge(fit_intercept=False).fit(scaledX.iloc[train], yy[train])
featureTotals += model.coef_
y_pred = model.predict(scaledX.iloc[test])
r2Total += r2_score(yy[test], y_pred); rmseTotal += np.sqrt(mean_squared_error(yy[test],
y_pred))
print(featureTotals / ([5] * scaledX.shape[1]))
else:
for train, test in tscv.split(scaledX, yy):
model = RandomForestRegressor(n_estimators=estimators,
random_state=42).fit(scaledX.iloc[train], yy[train])
y_pred = model.predict(scaledX.iloc[test])
r2Total += r2_score(yy[test], y_pred); rmseTotal += np.sqrt(mean_squared_error(yy[test],
y_pred))

return r2Total / 5, rmseTotal / 5
```

```
def seasonalityPredictor(y, q, d, w, lag, modelType='Ridge', x=0, stride=1, estimators=20,
evl=False):
lngth = y.size - w - lag * w - q
XX = 0

scaledX=0
if type(x) == type(pd.DataFrame()):
XX = y[q : q + lngth : stride]
for i in range(1, lag):
X = y[i * w + q : i * w + q + lngth : stride]
XX = np.vstack((XX, X))
for i in range(0, lag):
X = y[i * d + q : i * d + q + lngth : stride]
XX = np.vstack((XX, X))
for i in range(0, lag):
X = y[i : i + lngth : stride]
XX = np.vstack((XX, X))
x = x.iloc[(lag - 1) * w + w + q : (lag - 1) * w + w + q + lngth : stride].copy()
for i, X in enumerate(XX):
x['BIKE HISTORY ' + str(i + 1)] = X
scaled_features = StandardScaler().fit_transform(x.values)
scaledX = pd.DataFrame(scaled_features, columns=x.columns)
else:
XX = y[q : q + lngth : stride]
for i in range(1, lag):
X = y[i * w + q : i * w + q + lngth : stride]
XX = np.column_stack((XX, X))
for i in range(0, lag):
X = y[i * d + q : i * d + q + lngth : stride]
XX = np.column_stack((XX, X))
for i in range(0, lag):
X = y[i : i + lngth : stride]
XX = np.column_stack((XX, X))
featureTotals = [0] * XX.shape[1]
XX = StandardScaler().fit_transform(XX)

yy = y[lag * w + w + q : lag * w + w + q + lngth : stride]

tscv = TimeSeriesSplit()
r2Total = 0
rmseTotal = 0
maeTotal = 0
if modelType == 'Ridge':
for train, test in tscv.split(XX, yy):
model = Ridge(fit_intercept=False).fit(XX[train], yy[train])
featureTotals += model.coef_
y_pred = model.predict(XX[test])
```

```
r2Total += r2_score(yy[test], y_pred); rmseTotal += np.sqrt(mean_squared_error(yy[test],
y_pred)); maeTotal += mean_absolute_error(yy[test], y_pred)
print(featureTotals / ([5] * XX.shape[1]))
else:
for train, test in tscv.split(scaledX, yy):
model = RandomForestRegressor(n_estimators=estimators,
random_state=42).fit(scaledX.iloc[train], yy[train])
y_pred = model.predict(scaledX.iloc[test])
r2Total += r2_score(yy[test], y_pred); rmseTotal += np.sqrt(mean_squared_error(yy[test],
y_pred)); maeTotal += mean_absolute_error(yy[test], y_pred)

if evl:
return r2Total / 5, rmseTotal / 5, maeTotal / 5
return r2Total / 5, rmseTotal / 5

def findUsefulFeatures(x, y, dt, modelType, x1=0, y1=0):
d = math.floor(24 * 60 * 60 / dt)
w = math.floor(7 * 24 * 60 * 60 / dt)
Q = [2, 6, 12]

if modelType == 'Ridge':
print(f"\nDoing a basic run to see what features are useful and what aren't, the features
currently are: {x.columns}")
print(f'Blessington')
fullPredictor(x, y, 2, 1, 3)
print(f'Kilmainham')
fullPredictor(x1, y1, 2, 1, 3)

x = x.drop(['Unnamed: 0', 'STATION ID', 'BIKE STANDS', 'STATUS', 'LATITUDE', 'LONGITUDE'],
axis=1)
x1 = x1.drop(['Unnamed: 0', 'STATION ID', 'BIKE STANDS', 'STATUS', 'LATITUDE', 'LONGITUDE'],
axis=1)

print(f"\nDoing a basic run to see what features are useful and what aren't, the features
currently are: {x.columns}")
print(f'Blessington')
fullPredictor(x, y, 2, 1, 3)
print(f'Kilmainham')
fullPredictor(x1, y1, 2, 1, 3)
print('\n')

polynomials = [1, 2, 3, 4]
r2s, rmses, r2s_kil, rmses_kil = [], [], [], []
for p in polynomials:
xPoly = PolynomialFeatures(p).fit_transform(x)
xPoly = pd.DataFrame(xPoly)
r2, rmse = fullPredictor(xPoly, y, 2, 1, 3)
```

```
r2s.append(r2); rmses.append(rmse)
xPoly = PolynomialFeatures(p).fit_transform(x1)
xPoly = pd.DataFrame(xPoly)
r2, rmse = fullPredictor(xPoly, y1, 2, 1, 3)
r2s_kil.append(r2); rmses_kil.append(rmse)
plotResults(polynomials, r2s, rmses, 'polynomials', 'Looking at the effect of polynomial feature
engineering for Blessington')
plotResults(polynomials, r2s_kil, rmses_kil, 'polynomials', 'Looking at the effect of polynomial
feature engineering for Kilmainham')
plt.show()

# Going to plot q vs r2 and q vs rmse, with two colors, one for just time series data, and one
with full data
r2s, r2sFull, rmses, rmsesFull = [], [], [], []
r2s_kil, r2sFull_kil, rmses_kil, rmsesFull_kil = [], [], [], []
for q in Q:
print(f'Predicting {q} steps ahead')
print(f'Blessington')
r2, rmse = predictor(y, q, 1, 3)
r2s.append(r2); rmses.append(rmse)
r2, rmse = fullPredictor(x, y, q, 1, 3)
r2sFull.append(r2); rmsesFull.append(rmse)

print(f'Kilmainham')
r2, rmse = predictor(y1, q, 1, 3)
r2s_kil.append(r2); rmses_kil.append(rmse)
r2, rmse = fullPredictor(x1, y1, q, 1, 3)
r2sFull_kil.append(r2); rmsesFull_kil.append(rmse)
fig, axs = plt.subplots(2, 2)
fig.suptitle(f'Comparison of time series only model vs timeseries with extra data predictor
using Ridge')
axs[0, 0].scatter(x=Q, y=r2s); axs[0, 0].scatter(x=Q, y=r2sFull); axs[0, 0].set(xlabel='q',
ylabel='R squared'); #axs[0, 0].legend(['Time Series', 'Full Dataset']);
axs[0, 0].set_title("Blessington Street")
axs[1, 0].scatter(x=Q, y=rmses); axs[1, 0].scatter(x=Q, y=rmsesFull); axs[1,
0].set(xlabel='q', ylabel='RMSE'); #axs[1, 0].legend(['Time Series', 'Full Dataset'])
axs[0, 1].scatter(x=Q, y=r2s_kil); axs[0, 1].scatter(x=Q, y=r2sFull_kil); axs[0,
1].set(xlabel='q', ylabel='R squared'); #axs[0, 1].legend(['Time Series', 'Full Dataset']);
axs[0, 1].set_title("Kilmainham Gaol")
axs[1, 1].scatter(x=Q, y=rmses_kil); axs[1, 1].scatter(x=Q, y=rmsesFull_kil); axs[1,
1].set(xlabel='q', ylabel='RMSE'); #axs[1, 1].legend(['Time Series', 'Full Dataset'])
fig.legend(['Time Series', 'Full Dataset'])
plt.show()

streets = ['Blessington Street', 'Kilmainham Gaol']

### Find best lag
```

```
lag = [1, 2, 3, 4, 5, 6]
r2s, rmses, r2s_kil, rmses_kil = [], [], [], []
for l in lag:
r2, rmse = predictor(y, q, 1, l)
r2s.append(r2); rmses.append(rmse)

r2, rmse = predictor(y1, q, 1, l)
r2s_kil.append(r2); rmses_kil.append(rmse)
plotResults(lag, r2s, rmses, 'lag', f'Effect of lag on Ridge Regression predictions for 
Blessington Street')
plotResults(lag, r2s_kil, rmses_kil, 'lag', f'Effect of lag on Ridge Regression predictions for 
Kilmainham Gaol')

### Find best stride
stride = [1, 2, 3, 4, 5, 6]
r2s, rmses, r2s_kil, rmses_kil = [], [], [], []
for s in stride:
r2, rmse = predictor(y, q, 1, 3, stride=s)
r2s.append(r2); rmses.append(rmse)

r2, rmse = predictor(y1, q, 1, 3, stride=s)
r2s_kil.append(r2); rmses_kil.append(rmse)
plotResults(stride, r2s, rmses, 'stride', f'Effect of stride on Ridge Regression predictions for 
Blessington Street')
plotResults(stride, r2s_kil, rmses_kil, 'stride', f'Effect of stride on Ridge Regression predictions 
for Kilmainham Gaol')
plt.show()

immediateR2, dailyR2, weeklyR2, allR2 = [], [], [], []
immediateR2_kil, dailyR2_kil, weeklyR2_kil, allR2_kil = [], [], [], []
immediateRMSE, dailyRMSE, weeklyRMSE, allRMSE = [], [], [], []
immediateRMSE_kil, dailyRMSE_kil, weeklyRMSE_kil, allRMSE_kil = [], [], [], []
seasonality = ['Immediate', 'Daily', 'Weekly', 'All']
for q in Q:
print('Models with only occupancy data')
print(f'Predicting {q} steps ahead')
print('Immediate Trend')
r2, rmse = predictor(y, q, 1, 3)
immediateR2.append(r2); immediateRMSE.append(rmse)
print('Daily Seasonality')
r2, rmse = predictor(y, q, d, 3)
dailyR2.append(r2); dailyRMSE.append(rmse)
print('Weekly Seasonality')
r2, rmse = predictor(y, q, w, 3)
weeklyR2.append(r2); weeklyRMSE.append(rmse)
print('All Together')
r2, rmse = seasonalityPredictor(y, q, d, w, 3)
```

```
allR2.append(r2); allRMSE.append(rmse)
print('\n')

print('Immediate Trend')
r2, rmse = predictor(y1, q, 1, 3)
immediateR2_kil.append(r2); immediateRMSE_kil.append(rmse)
print('Daily Seasonality')
r2, rmse = predictor(y1, q, d, 3)
dailyR2_kil.append(r2); dailyRMSE_kil.append(rmse)
print('Weekly Seasonality')
r2, rmse = predictor(y1, q, w, 3)
weeklyR2_kil.append(r2); weeklyRMSE_kil.append(rmse)
print('All Together')
r2, rmse = seasonalityPredictor(y1, q, d, w, 3)
allR2_kil.append(r2); allRMSE_kil.append(rmse)
print('\n')
fig, axs = plt.subplots(2, 2)
fig.suptitle(f'Comparison of using different seasonalities using Ridge Regression')
axs[0, 0].scatter(x=Q, y=immediateR2); axs[0, 0].scatter(x=Q, y=dailyR2); axs[0,
0].scatter(x=Q, y=weeklyR2); axs[0, 0].scatter(x=Q, y=allR2); axs[0, 0].set(xlabel='q',
ylabel='R squared'); #axs[0, 0].legend(seasonality);
axs[0, 0].set_title("Blessington Street")
axs[1, 0].scatter(x=Q, y=immediateRMSE); axs[1, 0].scatter(x=Q, y=dailyRMSE); axs[1,
0].scatter(x=Q, y=weeklyRMSE); axs[1, 0].scatter(x=Q, y=allRMSE); axs[1, 0].set(xlabel='q',
ylabel='RMSE'); #axs[1, 0].legend(seasonality)
axs[0, 1].scatter(x=Q, y=immediateR2_kil); axs[0, 1].scatter(x=Q, y=dailyR2_kil); axs[0,
1].scatter(x=Q, y=weeklyR2_kil); axs[0, 1].scatter(x=Q, y=allR2_kil); axs[0, 1].set(xlabel='q',
ylabel='R squared'); #axs[0, 1].legend(seasonality);
axs[0, 1].set_title("Kilmainham Gaol")
axs[1, 1].scatter(x=Q, y=immediateRMSE_kil); axs[1, 1].scatter(x=Q, y=dailyRMSE_kil);
axs[1, 1].scatter(x=Q, y=weeklyRMSE_kil); axs[1, 1].scatter(x=Q, y=allRMSE_kil); axs[1,
1].set(xlabel='q', ylabel='RMSE'); #axs[1, 1].legend(seasonality)
fig.legend(seasonality)
plt.show()
else:
reducedX = x.copy()
c = reducedX.columns
c = c.drop('STATION ID').copy()
reducedX = reducedX.drop(c, axis=1).copy()
r2s, r2sFull, rmses, rmsesFull = [], [], [], []
for q in Q:
r2, rmse = predictor(y, q, 1, 3, 'Forest', x=reducedX)
r2s.append(r2); rmses.append(rmse)
r2, rmse = predictor(y, q, 1, 3, 'Forest', x=x)
r2sFull.append(r2); rmsesFull.append(rmse)
fig, (ax1, ax2) = plt.subplots(2)
```

```
fig.suptitle('Comparison of time series only model vs timeseries with extra data predictor on
Random Forest Model')
ax1.scatter(x=Q, y=r2s); ax1.scatter(x=Q, y=r2sFull); ax1.set(xlabel='q', ylabel='R
squared'); #ax1.legend(['Time Series', 'Full Dataset'])
ax2.scatter(x=Q, y=rmses); ax2.scatter(x=Q, y=rmsesFull); ax2.set(xlabel='q',
ylabel='RMSE'); #ax2.legend(['Time Series', 'Full Dataset'])
fig.legend(['Time Series', 'Full Dataset'])
plt.show()

### Find best lag
lag = [1, 2, 3, 4, 5, 6]
r2s, rmses = [], []
for l in lag:
tmpX = reducedX.copy()
r2, rmse = fullPredictor(tmpX, y, q, 1, l, 'Forest')
r2s.append(r2); rmses.append(rmse)
plotResults(lag, r2s, rmses, 'lag', f'Effect of lag on on Random Forest Model predictions')

### Find best stride
stride = [1, 2, 3, 4, 5, 6]
r2s, rmses = [], []
for s in stride:
tmpX = reducedX.copy()
r2, rmse = fullPredictor(tmpX, y, q, 1, 1, 'Forest', stride=s)
r2s.append(r2); rmses.append(rmse)
plotResults(stride, r2s, rmses, 'stride', f'Effect of stride on on Random Forest Model
predictions')

### Find best no. estimators
estimators = [1, 5, 10, 20, 50]
r2s, rmses = [], []
for e in estimators:
tmpX = reducedX.copy()
r2, rmse = fullPredictor(tmpX, y, q, 1, 1, 'Forest', stride=3, estimators=e)
r2s.append(r2); rmses.append(rmse)
plotResults(estimators, r2s, rmses, 'estimators', 'Effect of estimators on Random Forest Model
predictions')
plt.show()

immediateR2, dailyR2, weeklyR2, allR2 = [], [], [], []
immediateRMSE, dailyRMSE, weeklyRMSE, allRMSE = [], [], [], []
seasonality = ['Immediate', 'Daily', 'Weekly', 'All']
for q in Q:
print('Models with only occupancy data')
print(f'Predicting {q} steps ahead')
print('Immedieate Trend')
tmpX = reducedX.copy()
```

```
r2, rmse = fullPredictor(tmpX, y, q, 1, 1, 'Forest', stride=3, estimators=20)
immediateR2.append(r2); immediateRMSE.append(rmse)
print('Daily Seasonality')
tmpX = reducedX.copy()
r2, rmse = fullPredictor(tmpX, y, q, d, 1, 'Forest', stride=3, estimators=20)
dailyR2.append(r2); dailyRMSE.append(rmse)
print('Weekly Seasonality')
tmpX = reducedX.copy()
r2, rmse = fullPredictor(tmpX, y, q, w, 1, 'Forest', stride=3, estimators=20)
weeklyR2.append(r2); weeklyRMSE.append(rmse)
print('All Together')
tmpX = reducedX.copy()
r2, rmse = seasonalityPredictor(y, q, d, w, 6, 'Forest', x=tmpX, stride=3, estimators=20)
allR2.append(r2); allRMSE.append(rmse)
print('\n')
fig, (ax1, ax2) = plt.subplots(2)
fig.suptitle('Comparison of using different seasonalities on Random Forest Model')
ax1.scatter(x=Q, y=immediateR2); ax1.scatter(x=Q, y=dailyR2); ax1.scatter(x=Q,
y=weeklyR2); ax1.scatter(x=Q, y=allR2); ax1.set(xlabel='q', ylabel='R squared');
#ax1.legend(seasonality)
ax2.scatter(x=Q, y=immediateRMSE); ax2.scatter(x=Q, y=dailyRMSE); ax2.scatter(x=Q,
y=weeklyRMSE); ax2.scatter(x=Q, y=allRMSE); ax2.set(xlabel='q', ylabel='RMSE');
#ax2.legend(seasonality)
fig.legend(seasonality)
plt.show()

def evalDummy(model, y, q, lag, stride):
XX = y[0 : y.size - q - lag * 1 : stride]
for i in range(1, lag):
X = y[i * 1 : y.size - q - (lag - i) * 1 : stride]
XX = np.column_stack((XX, X))
yy = y[lag * 1 + q :: stride]
tscv = TimeSeriesSplit()
XX = StandardScaler().fit_transform(XX)

r2Total = 0
rmseTotal = 0
maeTotal = 0

for train, test in tscv.split(XX, yy):
model.fit(XX[train], yy[train])
y_pred = model.predict(XX[test])
r2Total += r2_score(yy[test], y_pred); rmseTotal += np.sqrt(mean_squared_error(yy[test],
y_pred)); maeTotal += mean_absolute_error(yy[test], y_pred)
return r2Total / 5, rmseTotal / 5, maeTotal / 5

def testModels(x, y1, y2, combinedY, dt):
```

```
reducedX = x.copy()
c = reducedX.columns
c = c.drop('STATION ID').copy()
x = reducedX.drop(c, axis=1).copy()
Q = [2, 6, 12]

dummyR2, ridgeBr2, ridgeKr2, forestR2 = [], [], [], []
dummyRMSE, ridgeBrmse, ridgeKrmse, forestRMSE = [], [], [], []
dummyMAE, ridgeBmae, ridgeKmae, forestMAE = [], [], [], []
# When testing the models, we're going to use R squared, RMSE, MAE
q = 2
print(f'Evaluations for predicting {q * 5} minutes ahead')

### Dummy
dummyY = combinedY[q :: 1]
model = DummyRegressor(strategy='mean')
r2, rmse, mae = evalDummy(model, dummyY, 1, 3, 1)
dummyR2.append(r2); dummyRMSE.append(rmse); dummyMAE.append(mae)
print(f'\nDummy model: \nR2 = {r2}, RMSE = {rmse}, MAE = {mae}')

### Ridge Seasonality: 1 lag: 1 for Blessington and 6 for Kilmainham Stride: 2 and 1
r2, rmse, mae = predictor(y1, q, 1, 2, stride=6, evl=True, prnt=False)
print(f'\nRidge model on Blessington Street: \nR2 = {r2}, RMSE = {rmse}, MAE = {mae}')
ridgeBr2.append(r2); ridgeBrmse.append(rmse); ridgeBmae.append(mae)
r2, rmse, mae = predictor(y2, q, 1, 6, stride=1, evl=True, prnt=False)
ridgeKr2.append(r2); ridgeKrmse.append(rmse); ridgeKmae.append(mae)
print(f'\nRidge model: Kilmainham Gaol\nR2 = {r2}, RMSE = {rmse}, MAE = {mae}')

### Forest Seasonality: 1 Lag: 1 Stride: 3
r2, rmse, mae = predictor(combinedY, q, 1, 1, modelType='Forest', x=x, stride=3,
estimators=20, evl=True)
forestR2.append(r2); forestRMSE.append(rmse); forestMAE.append(mae)
print(f'\nRandom forest regressor: \nR2 = {r2}, RMSE = {rmse}, MAE = {mae}')

q = 6
print(f'Evaluations for predicting {q * 5} minutes ahead')

### Dummy
dummyY = combinedY[q :: 1]
model = DummyRegressor(strategy='mean')
r2, rmse, mae = evalDummy(model, dummyY, 1, 3, 1)
dummyR2.append(r2); dummyRMSE.append(rmse); dummyMAE.append(mae)
print(f'\nDummy model: \nR2 = {r2}, RMSE = {rmse}, MAE = {mae}')

### Ridge Seasonality: 1 lag: Stride: 
r2, rmse, mae = predictor(y1, q, 1, 2, stride=6, evl=True, prnt=False)
ridgeBr2.append(r2); ridgeBrmse.append(rmse); ridgeBmae.append(mae)
```

```python
print(f'\nRidge model on Blessington Street: \nR2 = {r2}, RMSE = {rmse}, MAE = {mae}')
r2, rmse, mae = predictor(y2, q, 1, 6, stride=1, evl=True, prnt=False)
ridgeKr2.append(r2); ridgeKrmse.append(rmse); ridgeKmae.append(mae)
print(f'\nRidge model: Kilmainham Gaol\nR2 = {r2}, RMSE = {rmse}, MAE = {mae}')

### Forest Seasonality: Lag: Stride: 
r2, rmse, mae = predictor(combinedY, q, 1, 1, modelType='Forest', x=x, stride=3,
estimators=20, evl=True)
forestR2.append(r2); forestRMSE.append(rmse); forestMAE.append(mae)
print(f'\nRandom forest regressor: \nR2 = {r2}, RMSE = {rmse}, MAE = {mae}')

q = 12
print(f'Evaluations for predicting {q * 5} minutes ahead')

### Dummy
dummyY = combinedY[q :: 1]
model = DummyRegressor(strategy='mean')
r2, rmse, mae = evalDummy(model, dummyY, 1, 3, 1)
dummyR2.append(r2); dummyRMSE.append(rmse); dummyMAE.append(mae)
print(f'\nDummy model: \nR2 = {r2}, RMSE = {rmse}, MAE = {mae}')

### Ridge Seasonality: 1 lag: Stride: 
r2, rmse, mae = predictor(y1, q, 1, 2, stride=6, evl=True, prnt=False)
ridgeBr2.append(r2); ridgeBrmse.append(rmse); ridgeBmae.append(mae)
print(f'\nRidge model on Blessington Street: \nR2 = {r2}, RMSE = {rmse}, MAE = {mae}')
r2, rmse, mae = predictor(y2, q, 1, 6, stride=1, evl=True, prnt=False)
ridgeKr2.append(r2); ridgeKrmse.append(rmse); ridgeKmae.append(mae)
print(f'\nRidge model: Kilmainham Gaol\nR2 = {r2}, RMSE = {rmse}, MAE = {mae}')

### Forest Seasonality: Lag: Stride: 
r2, rmse, mae = predictor(combinedY, q, 1, 1, modelType='Forest', x=x, stride=3,
estimators=20, evl=True)
forestR2.append(r2); forestRMSE.append(rmse); forestMAE.append(mae)
print(f'\nRandom forest regressor: \nR2 = {r2}, RMSE = {rmse}, MAE = {mae}')

models=['Dummy', 'Blessington Ridge', 'Kilmainham Ridge', 'Random Forest']
plt.scatter(x=Q, y=dummyR2); plt.scatter(x=Q, y=ridgeBr2); plt.scatter(x=Q, y=ridgeKr2);
plt.scatter(x=Q, y=forestR2); plt.xlabel('q'); plt.ylabel('R squared'); plt.title('R squared
Results'); plt.legend(models)
plt.show()
plt.scatter(x=Q, y=dummyRMSE); plt.scatter(x=Q, y=ridgeBrmse); plt.scatter(x=Q,
y=ridgeKrmse); plt.scatter(x=Q, y=forestRMSE); plt.xlabel('q'); plt.ylabel('RMSE');
plt.title('RMSE Results'); plt.legend(models)
plt.show()
plt.scatter(x=Q, y=dummyMAE); plt.scatter(x=Q, y=ridgeBmae); plt.scatter(x=Q,
y=ridgeKmae); plt.scatter(x=Q, y=forestMAE); plt.xlabel('q'); plt.ylabel('MAE'); plt.title('MAE
Results'); plt.legend(models)
```

```
plt.show()

reduceStations()
data = pd.read_csv('./data/bike_data.csv', parse_dates=['TIME'])
xExtraBless, _ = transformData(data, 'BLESSINGTON STREET')
yBless, _, dt = extractOccupancyOnly(data, 'BLESSINGTON STREET')

xExtraKil, combinedExtraX = transformData(data, 'KILMAINHAM GAOL')
yKil, combinedY, dt = extractOccupancyOnly(data, 'KILMAINHAM GAOL')

findUsefulFeatures(xExtraBless, yBless, dt, 'Ridge', x1=xExtraKil, y1=yKil)

findUsefulFeatures(combinedExtraX, combinedY, dt, 'Forest')

testModels(combinedExtraX, yBless, yKil, combinedY, dt)
```