

Predicting Distribution of Dublin Bikes

Con O'Leary; olaoghac@tcd.ie

August 21, 2021

1 Task

To evaluate the feasibility of predicting bike station occupancy 10min, 30mins and 1 hour in the future; to do this by studying two stations with different patterns of behaviour.

2 Evaluation

My findings are that R^{**2} evaluation is an appropriate method for measuring success in this task. The result of the evaluation of a model should be directly proportionate to the efficacy that model provides to a hypothetical user. When a model makes a prediction about the number of bikes that will be at a station, if that prediction is off by only one bike, the error probably causes only a fifth of the inconvenience to the user as a prediction that is off by five bikes. As such, the fact that R^{**2} evaluation measures how correct the variance of predictions are—and that variance accounts equally for the frequency of errors, as well as the magnitude of errors—means that errors in the model's predictions are represented proportionally to how much of an issue is presented to a hypothetical user.

3 Data procurement

There are considerable time gaps in *Dublinbikes 2020 Q1 usage data*. I recognised this when I made a provisional graph (see below) that showed the percent of bike occupancy for each station throughout the full duration of the data set.

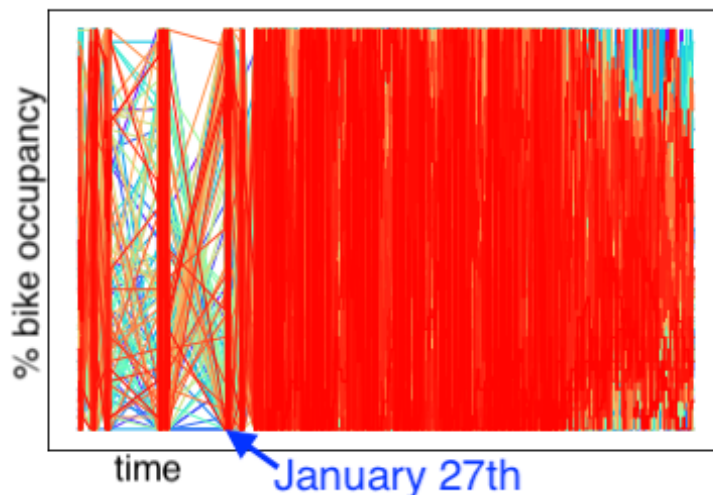


Figure 1: Lie graph representing the % bike occupancy for each station across the total duration (January 1st to April 1st)

The red is the line adjoining the x and y values of the final station to be plotted—as such it appears 'in front' of the lines representing the other stations. The sections of the x-axis that are atopped by solid colour is where data is present. Where there is a mesh of colours there is no data present. The mesh of colours is the graphing library adjoining a station's last data point before the dataless span to the station's first data point after the dataless span. The end of the second large dataless span is the 27th of January 2020; I used the information from then until the end of the dataset (April 1st). **Data was excluded before the 27th** to avoid a situation where the models would orientate more so around certain days of the week.

4 Baseline Approach

To ascertain how a baseline approach would perform at the task, I sought to develop a linear regression model. Noting the cyclical nature of the number of bikes in a station, I derived polynomial features from the amount of bikes in a given station over the total duration. I disabled the shuffle parameter in my train-test split, as it seemed testing would be inappropriate if, for example, we were looking for a prediction for the amount of bikes in station x in 30 minutes time when the prior training had included the amount of bikes in station x in 25 and 35 minutes time. Understandably, the x-axis of the training data spanning months and the x-axis of the testing data spanning weeks was not conducive of even nearly-positive R^2 scores, regardless of the degree of the polynomial features.

Next, with the same test/train division, I went about training 7 linear regressions—one for each day of the week. I trained the regressions independently, on their respective polynomial features, obtained from the share of data pertaining to that week day. When this approach fell through, I abandoned it; I felt spending more time to pursue fixes that added complexity to the approach would undermine the purpose of a baseline.

Finally, **I abandoned the use of models and established a baseline directly from the data:** the average bike occupancy per data point for each day of the week. The results of which were solid; scoring near-positive R^2 scores. Looking through a comparison of predicted and desired y-values I thought some form of regularisation would increase the accuracy. To regularise, I diluted my predictions with the mean y-value for the respective weekday of the given prediction. Optimising the coefficient that determines the degree to which the original predictions are diluted, I scored a marginally positive score for one of my stations, and a near-positive for the other. Interestingly this was the only approach that worked better for the more-central Custom House Quay station. Seeing as my baseline scored an R^2 score of roughly zero, this means its squared sum error was roughly that of the mean line of the training data. To gauge the efficacy of my baseline, as well as to see the effect of the regularisation, I plotted the R^2 scores of my baseline (with a range of values for my regularisation coefficient) versus the R^2 scores of the per-station mean of the training data. Realising the latter performed approximately as well as the approach I had developed, I adopted it as my baseline.

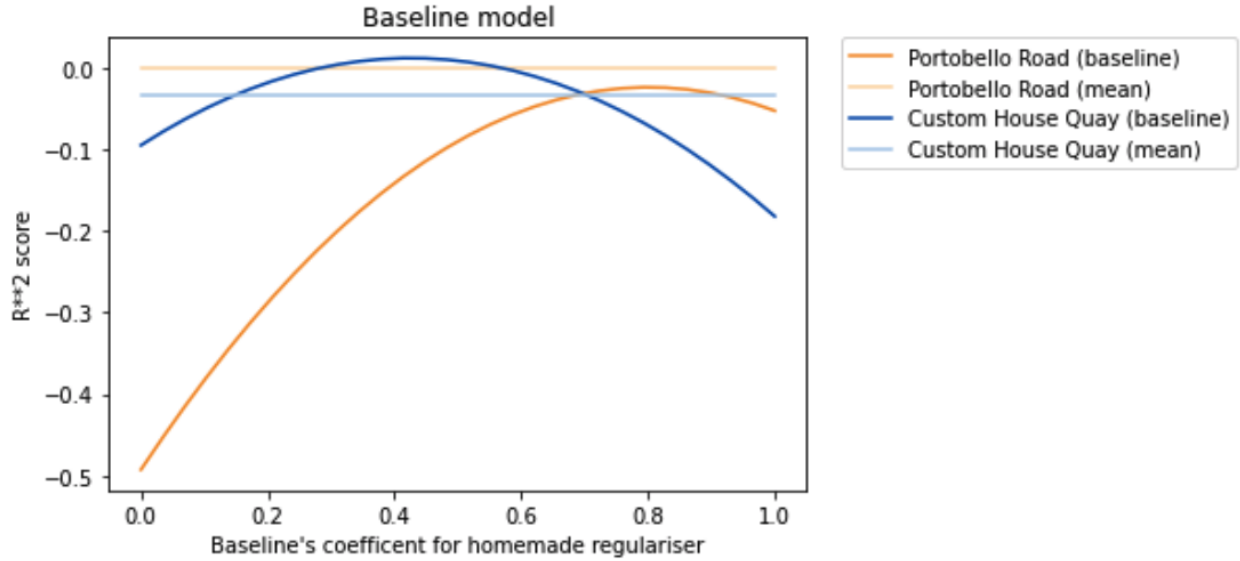


Figure 2: Optimising the regularisation of baseline predictions

5 Approach 1

The idea behind first approach was to identify patterns between bikes disappearing in stations and bikes turning up in other stations. As such I constructed a number of features: Chief among these is *bikes_changes_pastx*, which details for a given station at a given 5-minute interval the change in the number of bikes in the past x minutes. So that correlations between the changes detailed in *bikes_changes_pastx* could be found, time needed to be represented in features. The particular aspects of time that seemed to matter for the purposes of representing these patterns in bikes disappearing and turning up are the hour of day and the day of week. Figuring that an MLPRegressor might be a good means to represent these patterns, I made 7 inputs to represent the days of the week, and 23 to represent the time of the day. The 23 are represented as values between 0 and 1, such that 6,15 am would be represented by the 6th input having a value of 0.75, the 7th input having a value of 0.25, and the rest of the 23 inputs having a value of 0. Unlike this first time-related feature, I did not have the 7 inputs representing the days of the week transition gradually, but rather be 0 or 1: the degree of transition between the days is already represented by the first time-related feature. I also included a feature that is the percent fullness of the station—so that patterns between, for example, a station being quite empty and a large amount of bikes disappearing at a neighbouring station could be identified. Any features that did not already have values that ranged between 0 and 1 or -1 and 1 were normalised.

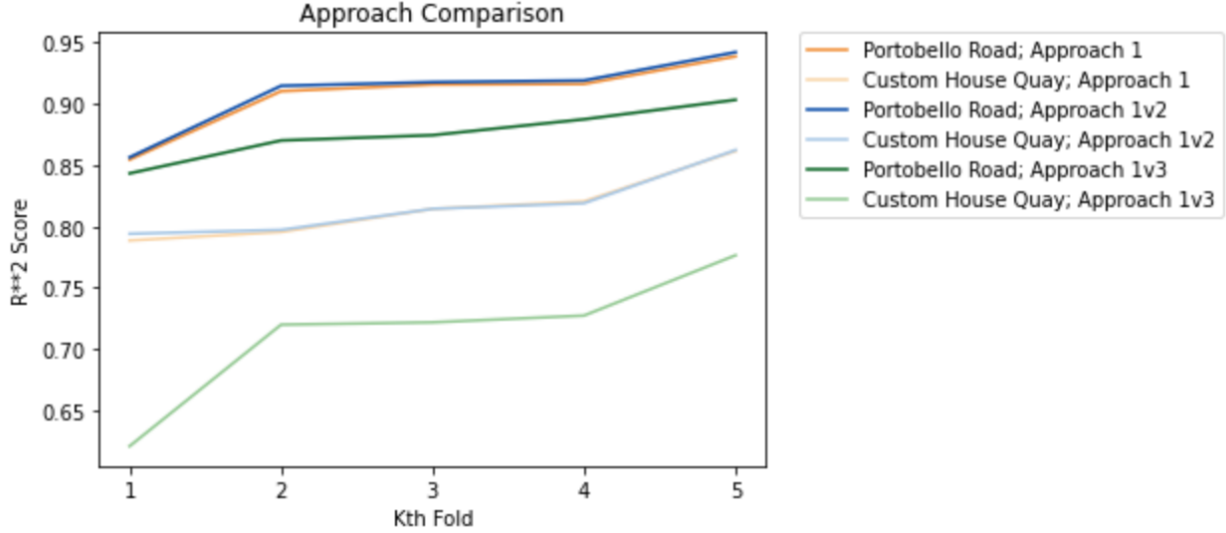


Figure 3: R**2 scores of variants of Approach 1

Variant 1 has the change in bike occupancy for the station in question over the past 5 and 10 minutes. Variant 2 is as Variant 1 but with the past 5, 10, 15, 20 and 25 minutes. Variant 3 has the change in bike occupancy for the station in question over the past 5 minutes, and the change in bike occupancy for all stations for the past 10, 15, 20 and 25 minutes.

While I had hoped the model would primarily derive its correlations from the various components of *bikes.changes.pastx*, instances of the model that did not have a whole kitchen sink of *bikes.changes.pastx* components thrown at them outperformed those that did. To be particular, the inclusion of (1) components pertaining to the change in amount of bikes over a duration greater than 10 minutes had no positive impact, and the inclusion of those greater than about 20 had a negative impact, and (2) the inclusion of components pertaining to the change in the amount of bikes in all stations had negative impacts. This was disappointing as I had hoped that a model having access to features that had the potential to describe inter-station bike transition patterns would discover such. Perhaps giving the changes in bike occupancy in all stations entailed too much irrelevant information. The way I would refine the model if I had more time would be to only look at bike changes in the past x minutes for stations that can be travelled to and from in x minutes. What gives further credence to the merit of the aforementioned adjustment is the disparity between the results for the two stations when Approach 1v3 is applied: This disparity shows that certain stations take fine to having the data for all stations, whereas others do not, which could indicate that optimising the conditions under which data is granted for other stations (e.g. proximity) could hold great potential.

6 Approach 2

My second approach orientates around finding recurring behaviour in a station's bike occupancy changes. To identify these recurrences I fit a KNN algorithm with features, such that the neighbours of a given datapoint for a given station are the datapoints for that station where the conditions of the features most closely resemble that of the datapoint in question. Like the first approach, it uses components from *bikes.changes.pastx*, as well as the stations percent occupancy. Time is represented differently to first approach. If the hour of day were to be represented as in the first approach, the KNN algorithm would regard 01:00 and 02:00 to be neighbourly, although it would not regard 23:00 and 00:00 to be neighbourly. As such, I represented the time of day as two different values, which are the xs and ys of a geometric circle. This means 23:00 and 00:00 would

be regarded by the algorithm as being as neighbourly as 01:00 and 02:00.

I think this approach took poorly to the components of *bikes_changes_pastx* pertaining to more than 10 minutes ago because, unlike the multi-layer perceptron model, the KNN algorithm does not weight coefficients for its features, so components pertaining to greater durations have the same weighting as the more-relevant past5 and past10 components.

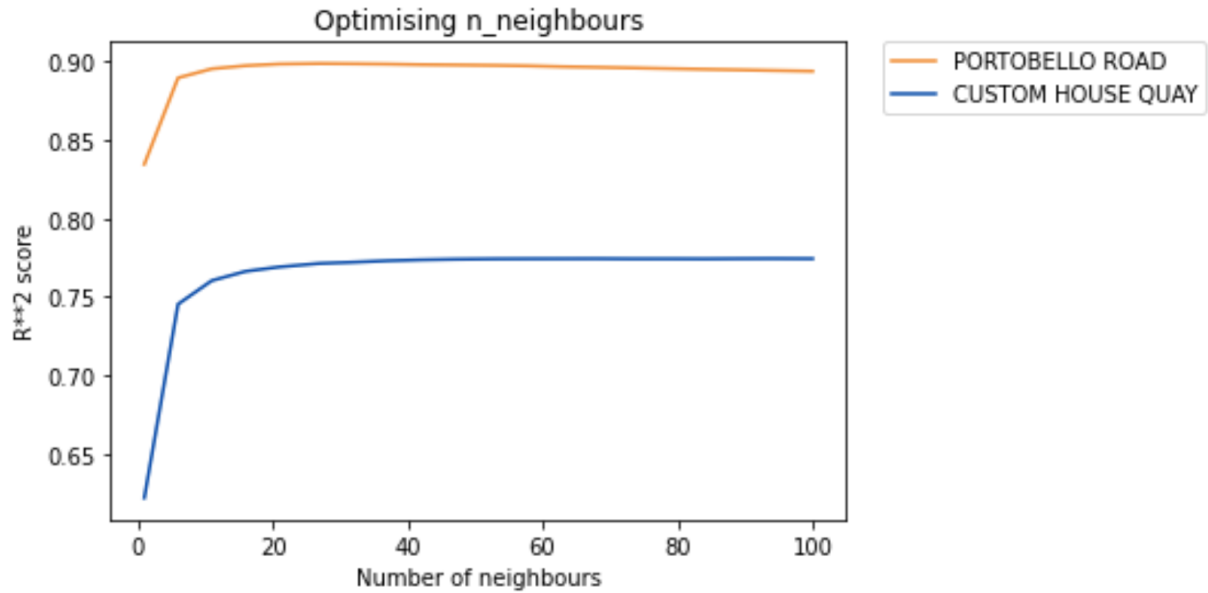


Figure 4: Optimising the number of neighbours

7 Conclusion

I was surprised that the KNN approach (approach 2) managed to perform nearly as well as the MLP approach.

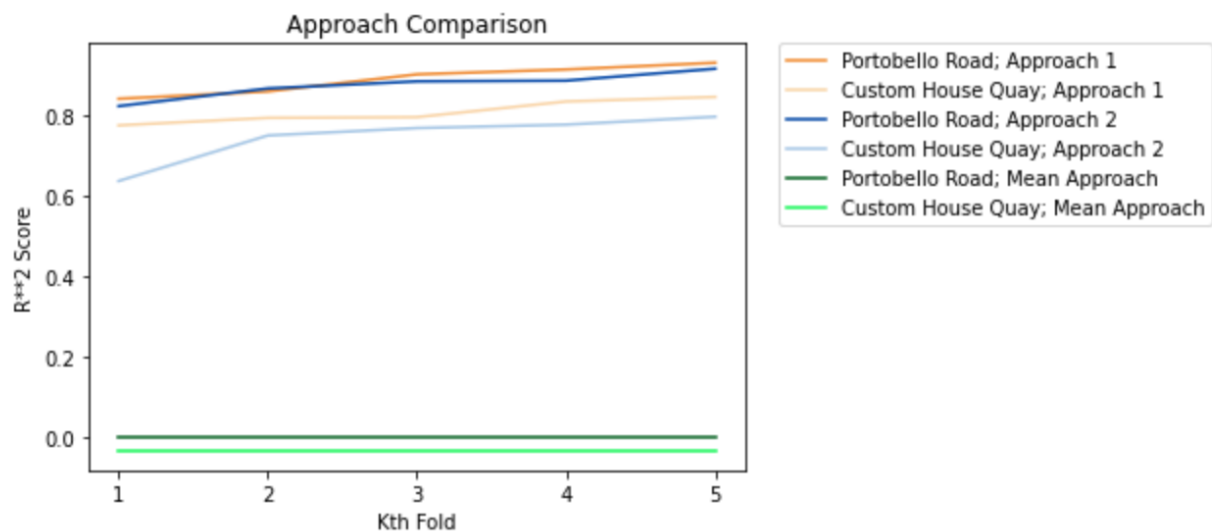


Figure 5: The two main approaches and the baseline approach

Either of the two approaches would probably suffice to drive a prediction feature for a Dublin Bikes app: the task does not require a particularly acute degree of accuracy, but rather consistency in being about right. The following graph plots the degree and severity of the errors of the two approaches when tasked to make the same predictions.

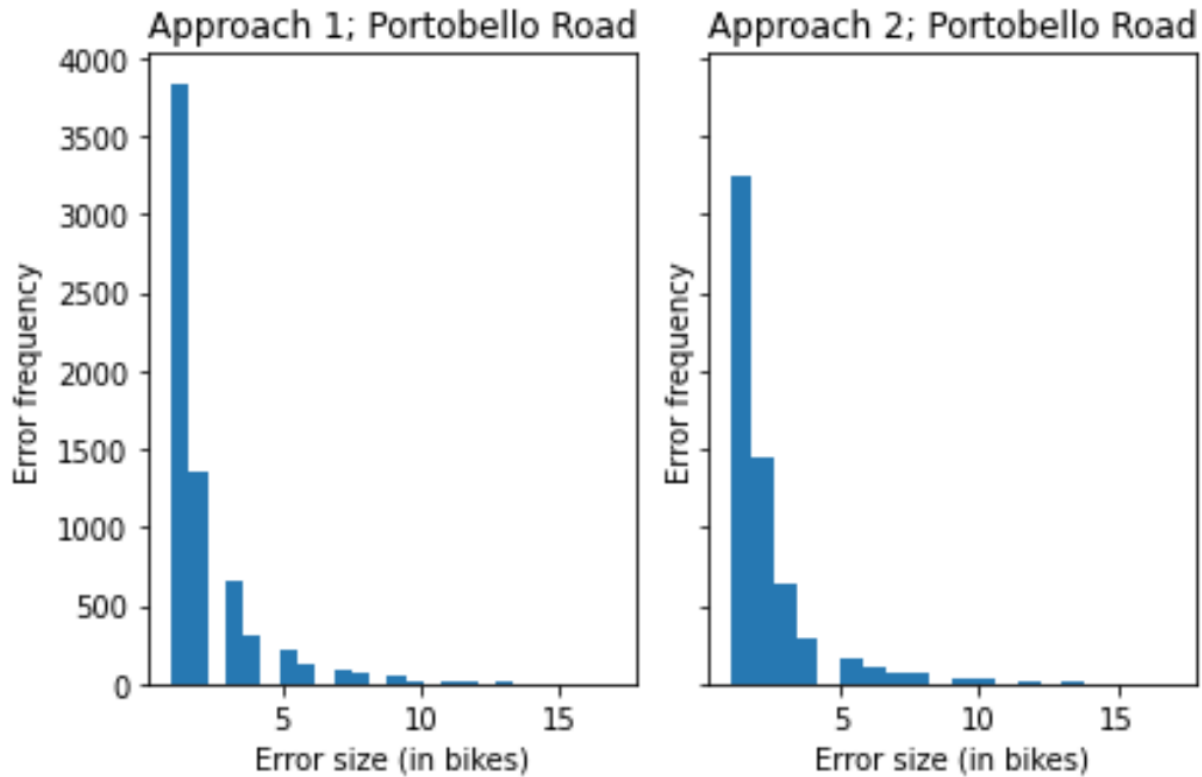


Figure 6: Of the erroneous predictions, the size and frequency of the error

In either case, it is immediately observable that roughly 90% of errors are off by between 1 and 4 bikes, which would be a nominal issues for users a great majority of the time. Nonetheless, I am certain R^2 scores in the high 90s could be achieved by a finer approach.

8 Code

```

1 # In[1]:
2
3
4 # IMPORTS & DEFINITIONS
5
6 import csv, sys
7 import datetime
8 import matplotlib.pyplot as plt
9 import matplotlib.cm as cm
10 import numpy as np; np.set_printoptions(threshold=sys.maxsize)
11 from sklearn.neural_network import MLPRegressor
12 from sklearn.datasets import make_regression
13 from sklearn.model_selection import train_test_split
14 from sklearn.model_selection import KFold
15 from sklearn.neighbors import KNeighborsRegressor
16 import math
17 from sklearn.model_selection import cross_val_score
18 from sklearn import linear_model
19 from sklearn.metrics import r2_score
20 from sklearn.preprocessing import PolynomialFeatures
21 from sklearn.metrics import mean_squared_error

```

```

22 from matplotlib.ticker import MaxNLocator
23 from sklearn.preprocessing import MinMaxScaler
24
25 DUD_VALUE= 0 # change from 0 to something like 123 for debugging
26 EMPTY_DATA_DAY_VAL= 123456789
27 TOTAL_ROWS= 999999999
28 INPUT_ROWS_LIMIT= TOTAL_ROWS # 500000
29 FILENAME= 'dublinbikes_2020_Q1.csv'
30 MAX_STATIONS= 118
31 SECS_IN_5MIN= 300
32 DATAPOINT_EVERYX_MIN= 5
33 DATAPOINTS_PER_DAY= 288
34 DAYS_OF_WEEK= ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday',
35                'Sunday'] # yes, I consider Monday to be the '0'/start of the week
36 STARTING_DATE= 0 # aka Monday. Because the 27th of Jan 2020 is a Monday
37 MISSING_STATIONS= [117, 116, 70, 60, 46, 35, 20, 14, 1, 0]
38 NUM_STATIONS= MAX_STATIONS - len(MISSING_STATIONS)
39 SUBSTANDARD_DAYS= [] # [50, 49]
40 TOTAL_DAYS= 66 # from 27 / 1 / 2020 to (and including) 1 / 4 / 2020
41 HOURS= 24
42 EPOCH= datetime.datetime(2020, 1, 27, 0, 0)
43 TOTAL_TIME_DATAPOINTS= int(((datetime.datetime(2020,4,2,0,0) - EPOCH).total_seconds
44                             () / SECS_IN_5MIN))
45 K= 5
46 STEP_SIZE= 0.02185 # just the magic number that leads to 288 values being generated
47 R= 0.5
48 MAX_HINDSIGHT= 60 # minutes
49 DAYS_PER_WEEKDAY= 5
50 HOMEMADE_REGULISER= 0.8
51 OPTIMAL_NEIGHBOURS= 30
52 MAX_ERROR_DIFF= 20
53
54 class DataDay: # ideally this would be nested in the Station class
55     def __init__(self, index):
56         self.index= index
57         self.substandard_day= False
58         if index in SUBSTANDARD_DAYS:
59             self.substandard_day= True
60         self.times_populated= 0
61         self.day_of_week= ((STARTING_DATE + index) % len(DAYS_OF_WEEK))
62
63         self.daily_epoch_time= np.full(DATAPOINTS_PER_DAY, EMPTY_DATA_DAY_VAL,
64                                         dtype=np.int)
65         self.epoch_time= np.full(DATAPOINTS_PER_DAY, EMPTY_DATA_DAY_VAL, dtype=np.
66                                 int)
67         self.bikes= np.full(DATAPOINTS_PER_DAY, EMPTY_DATA_DAY_VAL, dtype=np.int)
68         self.percent_bikes= np.full(DATAPOINTS_PER_DAY, float(EMPTY_DATA_DAY_VAL),
69                                     dtype=np.float)
70
71     def populate(self, daily_epoch_time, epoch_time, bikes, percent_bikes):
72         if self.substandard_day == False:
73             self.daily_epoch_time[daily_epoch_time]= daily_epoch_time
74             self.epoch_time[daily_epoch_time]= epoch_time
75             self.bikes[daily_epoch_time]= bikes
76             self.percent_bikes[daily_epoch_time]= percent_bikes
77             self.times_populated+= 1
78
79 class Station:
80     def __init__(self, index):
81         self.index= index
82         self.name= DUD_VALUE
83         self.bike_capacity= DUD_VALUE
84         self.address= DUD_VALUE
85         self.latitude= DUD_VALUE
86         self.longitude= DUD_VALUE
87         self.data_days= [DataDay(i) for i in range(0, TOTAL_DAYS)]
88

```



```

84     def populate_consts(self, name, bike_capacity, address, latitude, longitude):
85         self.name= name
86         self.bike_capacity= bike_capacity
87         self.address= address
88         self.latitude= latitude
89         self.longitude= longitude
90
91     def get_station_id(name):
92         try:
93             index= [x.name for x in stations].index(name)
94         except ValueError:
95             index= -1
96         return index
97
98
99 # In[2]:
100
101
102 # DATA STRUCTURING
103
104 total_capacity= 0 # not in use currently
105 index= []; daily_epoch_time= []; epoch_time= []; percent_bikes= [];
106 stations= [Station(i) for i in range(0, MAX_STATIONS)]
107 indices_to_populate= list(range(0, MAX_STATIONS))
108 for index in MISSING_STATIONS:
109     indices_to_populate.remove(index)
110
111 with open(FILENAME, newline='') as f:
112     reader = csv.reader(f); next(reader) # skip data header
113     current_index= 0
114     try:
115         while len(indices_to_populate) != 0:
116             row= next(reader)
117             if int(row[0]) == current_index: # this clause is just for performance
118                 continue
119             current_index= int(row[0])
120             if current_index in indices_to_populate:
121                 stations[current_index].populate_consts(row[3], row[4], row[8], row
122 [9], row[10])
123                 indices_to_populate.remove(current_index)
124                 total_capacity+= int(row[4])
125
126             f.seek(0)
127             reader= csv.reader(f); row= next(reader) # skip data header
128             for row_i, row in enumerate(reader):
129                 if row_i >= INPUT_ROWS_LIMIT:
130                     break
131                 if int(datetime.datetime(int(row[1][0:4]), int(row[1][5:7]), int(row
132 [1][8:10]), int(row[1][11: 13]), int(row[1][14: 16])) - EPOCH).total_seconds())
133 < 0:
134                     continue
135                 try:
136                     epoch_time= int((datetime.datetime(int(row[1][0:4]), int(row
137 [1][5:7]), int(row[1][8:10]), int(row[1][11: 13]), int(row[1][14: 16])) - EPOCH
138 ).total_seconds() / SECS_IN_5MIN)
139                     stations[int(row[0])].data_days[int(epoch_time / DATAPOINTS_PER_DAY
140 )].populate(
141                         int(datetime.datetime(int(row[1][0:4]), int(
142 row[1][5:7]), int(row[1][8:10]), int(row[1][11: 13]), int(row[1][14: 16])) -
143 datetime.datetime(int(row[1][0:4]), int(row[1][5:7]), int(row[1][8:10]), 0, 0))
144 .total_seconds() / (SECS_IN_5MIN)),
145                         epoch_time,
146                         int(row[6]),
147                         float("{:.3f}".format(float(row[6]) /
148 float(row[4])))
149                     except IndexError as e:
150                         print("Error:", e, int(row[0]))
151                         #print("\nTRIED: ", epoch_time, ' / ', DATAPOINTS_PER_DAY, ' = ',
152 int(epoch_time / DATAPOINTS_PER_DAY))
153                         #print(row[1])

```

```

139     except csv.Error as e:
140         sys.exit('file {}, line {}: {}'.format(filename, reader.line_num, e))
141
142 for station_i, station in enumerate(stations):
143     last_bikes= 0
144     last_percent_bikes= 0
145     for day_i, data_day in enumerate(station.data_days):
146         for val_i, val in enumerate(data_day.bikes):
147             if val == EMPTY_DATA_DAY_VAL:
148                 stations[station_i].data_days[day_i].populate(val_i, day_i *
DATAPOINTS_PER_DAY + val_i, last_bikes, last_percent_bikes)
149             else:
150                 last_bikes= data_day.bikes[val_i]
151                 last_percent_bikes= data_day.percent_bikes[val_i]
152
153
154 # In[3]:
155
156 # FEATURE DATA PREPERATION
157
158 fullness_in10= np.full((TOTAL_TIME_DATAPOINTS, NUM_STATIONS), DUD_VALUE, dtype=np.
int)
159 fullness_in30= np.full((TOTAL_TIME_DATAPOINTS, NUM_STATIONS), DUD_VALUE, dtype=np.
int)
160 fullness_in60= np.full((TOTAL_TIME_DATAPOINTS, NUM_STATIONS), DUD_VALUE, dtype=np.
int)
161 fullness= np.full((TOTAL_TIME_DATAPOINTS, NUM_STATIONS), DUD_VALUE, dtype=np.int)
162
163 fullness_percent= np.full((TOTAL_TIME_DATAPOINTS, NUM_STATIONS), DUD_VALUE, dtype=
np.float)
164
165 bikes_changes_pastx= np.full((TOTAL_TIME_DATAPOINTS, NUM_STATIONS, int(
MAX_HINDSIGHT / DATAPOINT_EVERYX_MIN)), DUD_VALUE, dtype=np.int)
166 days_of_week= np.full((TOTAL_TIME_DATAPOINTS, len(DAYS_OF_WEEK)), DUD_VALUE, dtype=
np.int)
167
168 hour_of_day= np.full((TOTAL_TIME_DATAPOINTS, HOURS), DUD_VALUE, dtype=np.float)
169 average_weekday_fullness= np.full((DATAPOINTS_PER_DAY, NUM_STATIONS, len(
DAYS_OF_WEEK)), DUD_VALUE, dtype=np.float)
170 weekdays_vol= np.full((NUM_STATIONS, len(DAYS_OF_WEEK)), 0, dtype=np.float)
171 avrg_weekday_full= np.full((NUM_STATIONS, len(DAYS_OF_WEEK)), 0, dtype=np.float)
172 meanmean= np.full((NUM_STATIONS, 0), dtype=np.float)
173
174 sclid_fullness_percent= np.full((TOTAL_TIME_DATAPOINTS, NUM_STATIONS), DUD_VALUE,
dtype=np.float)
175
176 sclid_bikes_changes_pastx= np.full((TOTAL_TIME_DATAPOINTS, NUM_STATIONS, int(
MAX_HINDSIGHT / DATAPOINT_EVERYX_MIN)), 0, dtype=np.float)
177
178 sclid_days_of_week= np.full((TOTAL_TIME_DATAPOINTS, len(DAYS_OF_WEEK)), DUD_VALUE,
dtype=np.int)
179
180 sclid_hour_of_week= np.full((TOTAL_TIME_DATAPOINTS, HOURS), DUD_VALUE, dtype=np.
float)
181
182
183 station_index_decrement= 0 # this is a varying offset for the indexing of stations
that accounts for missing stations that are being ignored
184
185 for epoch_day_i in range(TOTAL_DAYS):
186     #print("##### epoch_day_i: ", epoch_day_i)
187     x_offset= epoch_day_i * DATAPOINTS_PER_DAY
188     y_offset= 0
189
190     block= np.zeros((DATAPOINTS_PER_DAY, HOURS), dtype=np.float)
191     daily_epoch_time= list(range(DATAPOINTS_PER_DAY))
192     for time_i in daily_epoch_time:
193         hour= float("{:.3f}".format(time_i / 12)) # divide by 12 because there are
12 datapoints in an hour
194         block[time_i][(int(hour) + 1) % HOURS]= hour % 1
195         block[time_i][int(hour)]= 1 - (hour % 1)
196     hour_of_day[x_offset:x_offset + block.shape[0], y_offset:y_offset + block.shape
[1]]= block

```

```

191 day_of_week= stations[2].data_days[epoch_day_i].day_of_week
192 block= np.zeros((DATAPOINTS_PER_DAY, len(DAYS_OF_WEEK)), dtype=np.int)
193 for block_i, sub_arr in enumerate(block):
194     block[block_i][day_of_week]= 1
195 days_of_week[x_offset:x_offset + block.shape[0], y_offset:y_offset + block.
196 shape[1]]= block
197
198 for station in stations:
199     #print("##### station.index: ", station.index)
200     if station.index == 0:
201         station_index_decrement= 0
202     if station.index in MISSING_STATIONS:
203         station_index_decrement+= 1
204         continue
205     y_offset= station.index - station_index_decrement
206
207     block= station.data_days[epoch_day_i].percent_bikes
208     block= np.reshape(block, (DATAPOINTS_PER_DAY, 1))
209     fullness_percent[x_offset:x_offset + block.shape[0], y_offset:y_offset +
210 block.shape[1]]= block
211
212     block= station.data_days[epoch_day_i].bikes
213     block= np.reshape(block, (DATAPOINTS_PER_DAY, 1))
214     fullness[x_offset:x_offset + block.shape[0], y_offset:y_offset + block.
215 shape[1]]= block
216     block= np.reshape(block, (DATAPOINTS_PER_DAY, 1, 1))
217     if weekdays_vol[y_offset, day_of_week] < DAYS_PER_WEEKDAY:
218         average_weekday_fullness[0:DATAPOINTS_PER_DAY, y_offset:y_offset +
219 block.shape[1], day_of_week:day_of_week+1]+= block
220         weekdays_vol[y_offset:y_offset+1, day_of_week:day_of_week+1]+= 1
221
222     bikes= station.data_days[epoch_day_i].bikes
223     block= np.reshape(bikes[2:], (bikes.shape[0] - 2, 1))
224     fullness_in10[x_offset:x_offset + block.shape[0], y_offset:y_offset + block
225 .shape[1]]= block
226     block= np.reshape(bikes[6:], (bikes.shape[0] - 6, 1))
227     fullness_in30[x_offset:x_offset + block.shape[0], y_offset:y_offset + block
228 .shape[1]]= block
229     block= np.reshape(bikes[12:], (bikes.shape[0] - 12, 1))
230     fullness_in60[x_offset:x_offset + block.shape[0], y_offset:y_offset + block
231 .shape[1]]= block
232
233     block= np.reshape(station.data_days[epoch_day_i].bikes, (DATAPOINTS_PER_DAY
234 , 1))
235     if epoch_day_i - 1 == -1:
236         prev_block= np.zeros((DATAPOINTS_PER_DAY, 1), dtype=np.int)
237     else:
238         prev_block= np.reshape(station.data_days[epoch_day_i - 1].bikes, (
239 DATAPOINTS_PER_DAY, 1))
240     block_xminchange= np.zeros((DATAPOINTS_PER_DAY, int(MAX_HINDSIGHT /
241 DATAPOINT_EVERYX_MIN)), dtype=np.int)
242     fullness_xago= np.zeros((DATAPOINTS_PER_DAY, int(MAX_HINDSIGHT /
243 DATAPOINT_EVERYX_MIN)), dtype=np.int)
244     for col_i in range(fullness_xago.shape[1]):
245         i= col_i + 1
246         fullness_xago[i:DATAPOINTS_PER_DAY, col_i:col_i + 1]= block[0:
247 DATAPOINTS_PER_DAY - i, 0:1]
248         fullness_xago[0:i, col_i:col_i + 1]= prev_block[DATAPOINTS_PER_DAY - i:
249 DATAPOINTS_PER_DAY, 0:1]
250     for col_i in range(fullness_xago.shape[1]):
251         block_xminchange[0:DATAPOINTS_PER_DAY, col_i:col_i + 1]= np.subtract(
252 block, fullness_xago[0:DATAPOINTS_PER_DAY, col_i:col_i + 1])
253
254     bikes_changes_pastx[x_offset:x_offset + block_xminchange.shape[0], y_offset
255 :y_offset + 1, 0:block_xminchange.shape[1]]= np.reshape(block_xminchange, (
256 DATAPOINTS_PER_DAY, 1, block_xminchange.shape[1]))

```

```

242
243 station_index_decrement= 0 # this is a varying offset for the indexing of stations
    that accounts for missing stations that are being ignored
244 for station in stations:
245     if station.index == 0: # [117, 116, 70, 60, 46, 35, 20, 14, 1, 0]
246         station_index_decrement= 0
247     if station.index in MISSING_STATIONS:
248         station_index_decrement+= 1
249     continue
250 y_offset= station.index - station_index_decrement
251 for day_of_week_i in range(len(DAYS_OF_WEEK)):
252     average_weekday_fullness[0:DATAPOINTS_PER_DAY, y_offset:y_offset+1,
    day_of_week_i:day_of_week_i+1]/= weekdays_vol[y_offset:y_offset+1,
    day_of_week_i:day_of_week_i+1]
253     avrg_weekday_full[y_offset:y_offset+1, day_of_week_i:day_of_week_i+1]= np.
    mean(average_weekday_fullness[0:DATAPOINTS_PER_DAY, y_offset:y_offset+1,
    day_of_week_i:day_of_week_i+1])
254     meanmean[y_offset:y_offset+1]= np.mean(avrg_weekday_full[y_offset:y_offset+1])
255
256 # sld_fullness_percent= np.full((TOTAL_TIME_DATAPOINTS, NUM_STATIONS), DUD_VALUE,
    dtype=np.float)
257 # sld_bikes_changes_pastx= np.full((TOTAL_TIME_DATAPOINTS, NUM_STATIONS, int(
    MAX_HINDSIGHT / DATAPOINT_EVERYX_MIN)), 0, dtype=np.float)
258 # sld_hour_of_week= np.full((TOTAL_TIME_DATAPOINTS, HOURS), DUD_VALUE, dtype=np.
    float)
259
260 #####
261 for station_i in range(bikes_changes_pastx.shape[1]):
262     #print("##### STATION")
263     station_fullness= np.reshape(fullness_percent[0:TOTAL_TIME_DATAPOINTS,
    station_i:station_i+1], TOTAL_TIME_DATAPOINTS)
264     one_column= station_fullness.reshape(-1, 1)
265     scaler= MinMaxScaler((0, 1)).fit(one_column)
266     one_column= scaler.transform(one_column)
267     station_fullness= np.reshape(one_column, (station_fullness.shape[0], 1))
268     sld_fullness_percent[0:TOTAL_TIME_DATAPOINTS, station_i:station_i+1]=
    station_fullness
269
270     station_pastx= np.reshape(bikes_changes_pastx[0:TOTAL_TIME_DATAPOINTS,
    station_i:station_i+1, 0:bikes_changes_pastx.shape[2]], (TOTAL_TIME_DATAPOINTS,
    bikes_changes_pastx.shape[2]))
271     one_column= station_pastx.reshape(-1, 1)
272     scaler= MinMaxScaler((-1, 1)).fit(one_column)
273     one_column= scaler.transform(one_column)
274     station_pastx= np.reshape(one_column, (station_pastx.shape[0], 1, station_pastx
    .shape[1]))
275     sld_bikes_changes_pastx[0:TOTAL_TIME_DATAPOINTS, station_i:station_i+1, 0:
    bikes_changes_pastx.shape[2]]= station_pastx
276
277
278 # In[4]:
279
280
281 # APPROACH DEFINITIONS
282
283 errors= np.zeros((3, MAX_ERROR_DIFF), dtype=np.int)
284
285 def get_error(y_pred, y_test):
286     error= np.zeros(MAX_ERROR_DIFF, dtype=np.int)
287     for y_i, y in enumerate(y_pred):
288         for e_i in range(len(y_pred[y_i])):
289             val= y_pred[y_i][e_i]
290             val2= y_test[y_i][e_i]
291             diff= min(abs(round(val) - round(val2)), len(y_pred))
292             if diff != 0:
293                 error[diff - 1]+= 1
294     if not np.any(errors[0]):

```

```

295     errors[0]= error
296 elif not np.any(errors[1]):
297     errors[1]= error
298 elif not np.any(errors[2]):
299     errors[2]= error
300 else:
301     print("errors array all full!")
302
303 def run_approach1(station_name):
304     index= get_station_id(station_name)
305
306     y= np.full((TOTAL_TIME_DATAPOINTS, 3), 0, dtype=np.int) # change the 3 to a 6
307     # to do both stations at once on the generalised-training form of an approach
308     y[0:TOTAL_TIME_DATAPOINTS, 0:1]= np.reshape(fullness_in10[:,index], (
309     TOTAL_TIME_DATAPOINTS, 1))
310     y[0:TOTAL_TIME_DATAPOINTS, 1:2]= np.reshape(fullness_in30[:,index], (
311     TOTAL_TIME_DATAPOINTS, 1))
312     y[0:TOTAL_TIME_DATAPOINTS, 2:3]= np.reshape(fullness_in60[:,index], (
313     TOTAL_TIME_DATAPOINTS, 1))
314
315     X= np.full((TOTAL_TIME_DATAPOINTS, hour_of_day.shape[1] + days_of_week.shape[1]
316     + 3 + 0 * NUM_STATIONS), 0, dtype=np.float)
317     X[0:TOTAL_TIME_DATAPOINTS, 0:7]= day_of_week
318     X[0:TOTAL_TIME_DATAPOINTS, 7:31]= hour_of_day
319     X[0:TOTAL_TIME_DATAPOINTS, 31:32]= scld_fullness_percent[0:
320     TOTAL_TIME_DATAPOINTS, index:index + 1]
321     X[0:TOTAL_TIME_DATAPOINTS, 32:33]= np.reshape((scld_bikes_changes_pastx[0:
322     TOTAL_TIME_DATAPOINTS, index:index + 1, 0:1]), (TOTAL_TIME_DATAPOINTS, 1)) #
323     past5
324     X[0:TOTAL_TIME_DATAPOINTS, 33:34]= np.reshape((scld_bikes_changes_pastx[0:
325     TOTAL_TIME_DATAPOINTS, index:index + 1, 1:2]), (TOTAL_TIME_DATAPOINTS, 1)) #
326     past10
327
328     kf= KFold(n_splits= K)
329     kf.get_n_splits(X)
330     score_sum= 0.0
331     i= 1
332     returns= []
333     for train_index, test_index in kf.split(X):
334         X_train, X_test= X[train_index], X[test_index]
335         y_train, y_test= y[train_index], y[test_index]
336         regr= MLPRegressor(random_state= 1, max_iter= 1000, alpha=0.001).fit(
337         X_train, y_train)
338         y_pred= regr.predict(X_test)
339         score_sum+= regr.score(X_test, y_test)
340         returns.append(regr.score(X_test, y_test))
341         #print("R**2 score of data split", i, ": ", regr.score(X_test, y_test))
342         i+= 1
343     get_error(y_pred, y_test)
344     #print("\nAVERAGE R**2 score: ", score_sum / K)
345     return returns
346
347 def run_approach1v2(station_name):
348     index= get_station_id(station_name)
349
350     y= np.full((TOTAL_TIME_DATAPOINTS, 3), 0, dtype=np.int) # change the 3 to a 6
351     # to do both stations at once on the generalised-training form of an approach
352     y[0:TOTAL_TIME_DATAPOINTS, 0:1]= np.reshape(fullness_in10[:,index], (
353     TOTAL_TIME_DATAPOINTS, 1))
354     y[0:TOTAL_TIME_DATAPOINTS, 1:2]= np.reshape(fullness_in30[:,index], (
355     TOTAL_TIME_DATAPOINTS, 1))
356     y[0:TOTAL_TIME_DATAPOINTS, 2:3]= np.reshape(fullness_in60[:,index], (
357     TOTAL_TIME_DATAPOINTS, 1))
358
359     X= np.full((TOTAL_TIME_DATAPOINTS, hour_of_day.shape[1] + days_of_week.shape[1]
360     + 6 + 0 * NUM_STATIONS), 0, dtype=np.float)
361     X[0:TOTAL_TIME_DATAPOINTS, 0:7]= day_of_week

```

```

346 X[0:TOTAL_TIME_DATAPOINTS, 7:31]= hour_of_day
347 X[0:TOTAL_TIME_DATAPOINTS, 31:32]= scld_fullness_percent[0:
TOTAL_TIME_DATAPOINTS, index:index + 1]
348 X[0:TOTAL_TIME_DATAPOINTS, 32:33]= np.reshape((scld_bikes_changes_pastx[0:
TOTAL_TIME_DATAPOINTS, index:index+1, 0:1]), (TOTAL_TIME_DATAPOINTS, 1)) #
past5
349 X[0:TOTAL_TIME_DATAPOINTS, 33:34]= np.reshape((scld_bikes_changes_pastx[0:
TOTAL_TIME_DATAPOINTS, index:index+1, 1:2]), (TOTAL_TIME_DATAPOINTS, 1)) #
past10
350 X[0:TOTAL_TIME_DATAPOINTS, 34:35]= np.reshape((scld_bikes_changes_pastx[0:
TOTAL_TIME_DATAPOINTS, index:index+1, 2:3]), (TOTAL_TIME_DATAPOINTS, 1)) #
past15
351 X[0:TOTAL_TIME_DATAPOINTS, 35:36]= np.reshape((scld_bikes_changes_pastx[0:
TOTAL_TIME_DATAPOINTS, index:index+1, 3:4]), (TOTAL_TIME_DATAPOINTS, 1)) #
past20
352 X[0:TOTAL_TIME_DATAPOINTS, 36:37]= np.reshape((scld_bikes_changes_pastx[0:
TOTAL_TIME_DATAPOINTS, index:index+1, 4:5]), (TOTAL_TIME_DATAPOINTS, 1)) #
past25
353
354 kf= KFold(n_splits= K)
355 kf.get_n_splits(X)
356 score_sum= 0.0
357 i= 1
358 returns= []
359 for train_index, test_index in kf.split(X):
360     X_train, X_test= X[train_index], X[test_index]
361     y_train, y_test= y[train_index], y[test_index]
362     regr= MLPRegressor(random_state= 1, max_iter= 1000, alpha=0.001).fit(
X_train, y_train)
363     y_pred= regr.predict(X_test)
364     score_sum+= regr.score(X_test, y_test)
365     returns.append(regr.score(X_test, y_test))
366     #print("R**2 score of data split", i, ": ", regr.score(X_test, y_test))
367     i+= 1
368 get_error(y_pred, y_test)
369 #print("\nAVERAGE R**2 score: ", score_sum / K)
370 return returns
371
372 def run_approach1v3(station_name):
373     index= get_station_id(station_name)
374
375     y= np.full((TOTAL_TIME_DATAPOINTS, 3), 0, dtype=np.int) # change the 3 to a 6
to do both stations at once on the generalised-training form of an approach
376     y[0:TOTAL_TIME_DATAPOINTS, 0:1]= np.reshape(fullness_in10[:,index], (
TOTAL_TIME_DATAPOINTS, 1))
377     y[0:TOTAL_TIME_DATAPOINTS, 1:2]= np.reshape(fullness_in30[:,index], (
TOTAL_TIME_DATAPOINTS, 1))
378     y[0:TOTAL_TIME_DATAPOINTS, 2:3]= np.reshape(fullness_in60[:,index], (
TOTAL_TIME_DATAPOINTS, 1))
379
380     X= np.full((TOTAL_TIME_DATAPOINTS, hour_of_day.shape[1] + days_of_week.shape[1]
+ 2 + 4 * NUM_STATIONS), 0, dtype=np.float)
381     X[0:TOTAL_TIME_DATAPOINTS, 0:7]= day_of_week
382     X[0:TOTAL_TIME_DATAPOINTS, 7:31]= hour_of_day
383     X[0:TOTAL_TIME_DATAPOINTS, 31:32]= scld_fullness_percent[0:
TOTAL_TIME_DATAPOINTS, index:index + 1]
384     X[0:TOTAL_TIME_DATAPOINTS, 32:33]= np.reshape((scld_bikes_changes_pastx[0:
TOTAL_TIME_DATAPOINTS, index:index + 1, 0:1]), (TOTAL_TIME_DATAPOINTS, 1)) #
past5
385     X[0:TOTAL_TIME_DATAPOINTS, 33:141]= np.reshape((scld_bikes_changes_pastx[0:
TOTAL_TIME_DATAPOINTS, 0:NUM_STATIONS, 1:2]), (TOTAL_TIME_DATAPOINTS,
NUM_STATIONS)) # past10
386     X[0:TOTAL_TIME_DATAPOINTS, 141:249]= np.reshape((scld_bikes_changes_pastx[0:
TOTAL_TIME_DATAPOINTS, 0:NUM_STATIONS, 2:3]), (TOTAL_TIME_DATAPOINTS,
NUM_STATIONS)) # past15
387     X[0:TOTAL_TIME_DATAPOINTS, 249:357]= np.reshape((scld_bikes_changes_pastx[0:
TOTAL_TIME_DATAPOINTS, 0:NUM_STATIONS, 3:4]), (TOTAL_TIME_DATAPOINTS,

```

```

388 NUM_STATIONS)) # past20
X[0:TOTAL_TIME_DATAPOINTS, 357:465]= np.reshape((sclد_bikes_changes_pastx[0:
TOTAL_TIME_DATAPOINTS, 0:NUM_STATIONS, 4:5]), (TOTAL_TIME_DATAPOINTS,
NUM_STATIONS)) # past25

389
390 kf= KFold(n_splits= K)
391 kf.get_n_splits(X)
392 score_sum= 0.0
393 i= 1
394 returns= []
395 for train_index, test_index in kf.split(X):
396     X_train, X_test= X[train_index], X[test_index]
397     y_train, y_test= y[train_index], y[test_index]
398     regr= MLPRegressor(random_state= 1, max_iter= 1000, alpha=0.001).fit(
X_train, y_train)
399     y_pred= regr.predict(X_test)
400     score_sum+= regr.score(X_test, y_test)
401     returns.append(regr.score(X_test, y_test))
402     #print("R**2 score of data split", i, ": ", regr.score(X_test, y_test))
403     i+= 1
404 get_error(y_pred, y_test)
405 #print("\nAVERAGE R**2 score: ", score_sum / K)
406 return returns
407
408 def run_approach2(station_name, neighs= OPTIMAL_NEIGHBOURS):
409     index= get_station_id(station_name)
410
411     y= np.full((TOTAL_TIME_DATAPOINTS, 3), 0, dtype=np.int) # change the 3 to a 6
412     # to do both stations at once on the generalised-training form of an approach
413     y[0:TOTAL_TIME_DATAPOINTS, 0:1]= np.reshape(fullness_in10[:,index], (
TOTAL_TIME_DATAPOINTS, 1))
414     y[0:TOTAL_TIME_DATAPOINTS, 1:2]= np.reshape(fullness_in30[:,index], (
TOTAL_TIME_DATAPOINTS, 1))
415     y[0:TOTAL_TIME_DATAPOINTS, 2:3]= np.reshape(fullness_in60[:,index], (
TOTAL_TIME_DATAPOINTS, 1))
416
417     X= np.full((TOTAL_TIME_DATAPOINTS, 2 + 3
418                 #* bikes_changes_pastx.
419                 shape[1] \ # This line is uncommented when training on all stations
420                 ), -1, dtype=np.float)
421
422     positions= []; t= 0
423     while t < 2 * math.pi:
424         positions.append((1 - (R * math.cos(t) + R), R * math.sin(t) + R))
425         t+= STEP_SIZE
426     pos_i= 0
427     for time_i in range(TOTAL_TIME_DATAPOINTS):
428         X[time_i, 0]= positions[pos_i][0]
429         X[time_i, 1]= positions[pos_i][1]
430         pos_i= (pos_i + 1) % len(positions)
431
432     X[0:TOTAL_TIME_DATAPOINTS, 2:3]= sclد_fullness_percent[0:TOTAL_TIME_DATAPOINTS,
index:index+1]
433     X[0:TOTAL_TIME_DATAPOINTS, 3:4]= np.reshape((sclд_bikes_changes_pastx[0:
TOTAL_TIME_DATAPOINTS, index:index+1, 0:1]), (TOTAL_TIME_DATAPOINTS, 1)) #
434     past5
435     X[0:TOTAL_TIME_DATAPOINTS, 4:5]= np.reshape((sclд_bikes_changes_pastx[0:
TOTAL_TIME_DATAPOINTS, index:index+1, 1:2]), (TOTAL_TIME_DATAPOINTS, 1)) #
436     past5
437     # X[0:TOTAL_TIME_DATAPOINTS, 2:110]= bikes_changes_past5
438     # X[0:TOTAL_TIME_DATAPOINTS, 110:218]= bikes_changes_past15
439
440     kf= KFold(n_splits= K)
441     kf.get_n_splits(X)
442     score_sum= 0.0
443     i= 1
444     returns= []
445     for train_index, test_index in kf.split(X):

```

```

441     X_train, X_test= X[train_index], X[test_index]
442     y_train, y_test= y[train_index], y[test_index]
443     neigh= KNeighborsRegressor(n_neighbors= neighs, weights='distance').fit(
X_train, y_train)
444     y_pred= neigh.predict(X_test)
445     score_sum+= neigh.score(X_test, y_test)
446     returns.append(neigh.score(X_test, y_test))
447     #print("R**2 score of data split", i, ": ", regr.score(X_test, y_test))
448     i+= 1
449     get_error(y_pred, y_test)
450     #print("\nAVERAGE R**2 score: ", score_sum / K)
451     return returns
452
453 def run_oldbaseline(station_name, regulariser_coef):
454     index= get_station_id(station_name)
455     max_train_time= DATAPOINTS_PER_DAY * DAYS_PER_WEEKDAY * len(DAYS_OF_WEEK)
456     y_test= np.reshape(fullness[max_train_time:TOTAL_TIME_DATAPOINTS, index:index
+1], TOTAL_TIME_DATAPOINTS - max_train_time)
457     y_pred= np.zeros(TOTAL_TIME_DATAPOINTS - max_train_time)
458     for i in range(int((TOTAL_TIME_DATAPOINTS - max_train_time) /
DATAPOINTS_PER_DAY)):
459         datapoint_i= i * DATAPOINTS_PER_DAY
460         day_of_week_i= int((max_train_time + datapoint_i) / DATAPOINTS_PER_DAY) %
len(DAYS_OF_WEEK)
461         y_pred[datapoint_i:datapoint_i + DATAPOINTS_PER_DAY]= (np.reshape(
average_weekday_fullness[0:DATAPOINTS_PER_DAY, index:index+1, day_of_week_i:
day_of_week_i+1], DATAPOINTS_PER_DAY) * (1 - regulariser_coef) + np.full(
DATAPOINTS_PER_DAY, avrg_weekday_full[index:index+1, day_of_week_i:
day_of_week_i+1]) * regulariser_coef)
462     #print("R**2 score: ", r2_score(y_test, y_pred))
463     return r2_score(y_test, y_pred)
464
465 def run_meanline(station_name):
466     index= get_station_id(station_name)
467     max_train_time= DATAPOINTS_PER_DAY * DAYS_PER_WEEKDAY * len(DAYS_OF_WEEK)
468     y_test= np.reshape(fullness[max_train_time:TOTAL_TIME_DATAPOINTS, index:index
+1], TOTAL_TIME_DATAPOINTS - max_train_time)
469     y_pred= np.zeros(TOTAL_TIME_DATAPOINTS - max_train_time)
470     for i in range(int((TOTAL_TIME_DATAPOINTS - max_train_time) /
DATAPOINTS_PER_DAY)):
471         datapoint_i= i * DATAPOINTS_PER_DAY
472         day_of_week_i= int((max_train_time + datapoint_i) / DATAPOINTS_PER_DAY) %
len(DAYS_OF_WEEK)
473         y_pred[datapoint_i:datapoint_i + DATAPOINTS_PER_DAY]= np.full(
DATAPOINTS_PER_DAY, meanmean[index:index+1], dtype=np.float64)
474     #print("R**2 score: ", r2_score(y_test, y_pred))
475     return r2_score(y_test, y_pred)
476
477
478 # In[7]:
479
480 def baseline_graph():
481     meanmean1= run_meanline("PORTOBELLO ROAD")
482     meanmean2= run_meanline("CUSTOM HOUSE QUAY")
483     coefs= np.linspace(0, 1, num=30)
484
485     s1_r2= []
486     s2_r2= []
487     s1_r2meanmean= []
488     s2_r2meanmean= []
489
490     for coef in coefs:
491         s1_r2.append(run_oldbaseline("PORTOBELLO ROAD", coef))
492         s2_r2.append(run_oldbaseline("CUSTOM HOUSE QUAY", coef))
493         s1_r2meanmean.append(meanmean1)
494         s2_r2meanmean.append(meanmean2)

```



```

496 ax= plt.gca()
497
498
499 ax.plot(coefs, s1_r2, label="Portobello Road (baseline)", color="#F28C28")
500 ax.plot(coefs, s1_r2meanmean, label="Portobello Road (mean)", color="#FAD5A5")
501 ax.plot(coefs, s2_r2, label="Custom House Quay (baseline)", color="#0047AB")
502 ax.plot(coefs, s2_r2meanmean, label="Custom House Quay (mean)", color="#A7C7E7"
503 )
504
505 plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
506
507 plt.xlabel('Baseline\'s coefficient for homemade regulariser')
508 plt.ylabel('R**2 score')
509 plt.title('Baseline model')
510 plt.show()
511
512 def neighbours_optimisation(station_name1, station_name2):
513     xs= [int(x) for x in np.linspace(1, 100, num=20)]
514     y1s= []; y2s= []
515     for x in xs:
516         returns= run_approach2(station_name1, x)
517         y1s.append(sum(returns) / len(returns))
518         returns= run_approach2(station_name2, x)
519         y2s.append(sum(returns) / len(returns))
520     ax= plt.gca()
521
522     ax.plot(xs, y1s, label=station_name1, color="#F28C28")
523     ax.plot(xs, y2s, label=station_name2, color="#0047AB")
524
525     plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
526
527     plt.xlabel('Number of neighbours')
528     plt.ylabel('R**2 score')
529     plt.title('Optimising n_neighbours')
530     plt.show()
531
532 def compare_approaches(station_name1, station_name2, approach1, approach2,
533 approach3=None):
534     s1a1_r2s= []; s2a1_r2s= []; s1a2_r2s= []; s2a2_r2s= []; s1a3_r2s= []; s2a3_r2s=
535     []
536
537     val= [0.8544876302212273, 0.9103206998138718, 0.9156765327292385,
538     0.9162074563960463, 0.9386270881532218]#approach1(station_name1)
539     #[0.9154489426476711, 0.9321981853574037, 0.9033862664158813,
540     0.8607531451151377, 0.8425975287920906]#approach1(station_name1)
541     print("approach1(station_name1):", val)
542     if type(val) is list:
543         s1a1_r2s= sorted(val)
544     else:
545         s1a1_r2s.append(val); s1a1_r2s.append(val); s1a1_r2s.append(val); s1a1_r2s.
546         append(val); s1a1_r2s.append(val)
547
548     val= [0.795526303673726, 0.8143441006746407, 0.8204766460302203,
549     0.8615393834469698, 0.788609683100724]#approach1(station_name2)
550     #[0.77629295945547, 0.7949464686296777, 0.7967860515294295, 0.8358203281148665,
551     0.8471048354650209]#approach1(station_name2)
552     print("approach1(station_name2):", val)
553     if type(val) is list:
554         s2a1_r2s= sorted(val)
555     else:
556         s2a1_r2s.append(val); s2a1_r2s.append(val); s2a1_r2s.append(val); s2a1_r2s.
557         append(val); s2a1_r2s.append(val)
558
559     #
560     #####
561
562     val= [0.9190271650242875, 0.9420657457918499, 0.9146759986093368,
563     0.917653317897591, 0.8564092920020631]#approach2(station_name1)

```

```

549     print("approach2(station_name1):", val)
550     if type(val) is list:
551         s1a2_r2s= sorted(val)
552     else:
553         s1a2_r2s.append(val); s1a2_r2s.append(val); s1a2_r2s.append(val); s1a2_r2s.
append(val); s1a2_r2s.append(val)
554
555     val= [0.7972544574987751, 0.8143952998700232, 0.8189551156559984,
0.8623204399472701, 0.7940865400569425]#approach2(station_name2)
556     print("approach2(station_name2):", val)
557     if type(val) is list:
558         s2a2_r2s= sorted(val)
559     else:
560         s2a2_r2s.append(val); s2a2_r2s.append(val); s2a2_r2s.append(val); s2a2_r2s.
append(val); s2a2_r2s.append(val)
561 #
#####
562     if approach3 != None:
563         val= [0.8701196589704495, 0.9032181011324892, 0.8744269239638829,
0.8873912710329473, 0.843289038452534]#approach3(station_name1)
564         print("approach3(station_name1):", val)
565         if type(val) is list:
566             s1a3_r2s= sorted(val)
567         else:
568             s1a3_r2s.append(val); s1a3_r2s.append(val); s1a3_r2s.append(val);
s1a3_r2s.append(val); s1a3_r2s.append(val)
569
570         val= [0.6209740089315421, 0.7218889672070009, 0.7274194072782662,
0.7765443329528768, 0.7200027086352144]#approach3(station_name2)
571         print("approach3(station_name2):", val)
572         if type(val) is list:
573             s2a3_r2s= sorted(val)
574         else:
575             s2a3_r2s.append(val); s2a3_r2s.append(val); s2a3_r2s.append(val);
s2a3_r2s.append(val); s2a3_r2s.append(val)
576
577     print("s1a1_r2s:", s1a1_r2s)
578     print("s2a1_r2s:", s2a1_r2s)
579     print("s1a2_r2s:", s1a2_r2s)
580     print("s2a2_r2s:", s2a2_r2s)
581     if approach3 != None:
582         print("s1a3_r2s:", s1a3_r2s)
583         print("s2a3_r2s:", s2a3_r2s)
584
585     x= np.linspace(1, 5, num=K, dtype=np.int)
586
587     ax= plt.gca()
588     ax.xaxis.set_major_locator(MaxNLocator(integer=True))
589
590     ax.plot(x, s1a1_r2s, label="Portobello Road; Approach 1", color="#F28C28")
591     ax.plot(x, s2a1_r2s, label="Custom House Quay; Approach 1", color="#FAD5A5")
592     ax.plot(x, s1a2_r2s, label="Portobello Road; Approach 1v2", color="#0047AB")
593     ax.plot(x, s2a2_r2s, label="Custom House Quay; Approach 1v2", color="#A7C7E7")
594     if approach3 != None:
595         ax.plot(x, s1a3_r2s, label="Portobello Road; Approach 1v3", color="#026420"
)
596         ax.plot(x, s2a3_r2s, label="Custom House Quay; Approach 1v3", color="#92
CA91")
597
598     plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
599
600     plt.xlabel('Kth Fold')
601     plt.ylabel('R**2 Score')
602     plt.title('Approach Comparison')
603     plt.show()
604

```

```

605 def error_histo():
606     x= []
607     x2= []
608     for e_i, e in enumerate(errors[0]):
609         for n in range(e.astype(int).item()):
610             x.append(e_i + 1)
611     for e_i, e in enumerate(errors[1]):
612         for n in range(e.astype(int).item()):
613             x2.append(e_i + 1)
614
615     n_bins= MAX_ERROR_DIFF
616
617     fig, axs= plt.subplots(1, 2, sharey=True, sharex=True, tight_layout=True)
618
619     # We can set the number of bins with the 'bins' kwarg
620     axs[0].hist(x, bins= n_bins)
621     axs[0].set_xlabel('Error size (in bikes)', ylabel='Error frequency')
622     axs[0].set_title('Approach 1; Portobello Road')
623     axs[1].hist(x2, bins= n_bins)
624     axs[1].set_xlabel('Error size (in bikes)', ylabel='Error frequency')
625     axs[1].set_title('Approach 2; Portobello Road')
626
627
628 # In[8]:
629
630
631 # DRIVER
632 errors= np.zeros((3, MAX_ERROR_DIFF), dtype=np.int)
633 run_approach1("PORTOBELLO ROAD")
634 run_approach2("PORTOBELLO ROAD")
635 error_histo()
636
637 # neighbours_optimisation("PORTOBELLO ROAD", "CUSTOM HOUSE QUAY")
638 print("-----")

```