CS156 Assignment 3

Akmarzhan Abylay

October 2020

Yosemite Village yearly weather

Instructions: Temperature is cyclical, not only on a 24 hour basis but also on a yearly basis. Convert the dataset into a richer format whereby the day of the year is also captured. For example the time "20150212 1605", can be converted into (43, 965) because the 12th of February is the 43rd day of the year, and 16:05 is the 965th minute of the day.

This data covers 6 years, so split the data into a training set of the first 5 years, and a testing set of the 6th year.

Data Preprocessing

Step 1: Load the data

Since we only need the day and time of the day (hour+minutes) data for the tasks, I only loaded them. I also renamed the column for more comfortable use.

Also, since we are working with the data from 2011 (beginning) to 2016 (end), I deleted the data points from 2017 (1 point from January 1st). We could also use it, but I decided not to. Although more is better in terms of the number of data points, since we already have many data points (600k+), deleting 1 point wouldn't make a big difference.

[1]:
```
import numpy as np
import pandas as pd
from sklearn.metrics.pairwise import rbf_kernel
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
import matplotlib.pyplot as plt
%matplotlib inline

years = range(2011, 2017)
files = ['CRNS0101-05-%d-CA_Yosemite_Village_12_W.txt' % y for y in years]
usecols = [1, 2, 8]

load = [np.loadtxt(f, usecols=usecols) for f in files]
load = np.vstack(load)
```

[2]:
```
data = pd.DataFrame(load)

#renaming
data.rename(columns={0: 'date',
                     1: 'time',
                     2: 'temp', }, inplace=True)
data.head()
```

[2]:

	date	time	temp
0	20110101.0	5.0	-6.4
1	20110101.0	10.0	-6.5
2	20110101.0	15.0	-6.5
3	20110101.0	20.0	-6.5
4	20110101.0	25.0	-6.7

Step 2: Cleaning

There are some weird temperatures (i.e., -9999), which I removed. You can see that I removed around 400 data points.

[3]:
```
print("Minimum temperature is:", min(data.temp)) #some weird temperature
print("What other unique values are weird?", np.unique(data.temp[data.temp <␣
↪-100])[0]) #only this one value is weird

print("\nOriginal length:", len(data))
data = data[data.temp >= -100]
print("New length:", len(data))
```

```
Minimum temperature is: -9999.0
What other unique values are weird? -9999.0

Original length: 631296
New length: 630854
```

Step 3: Convertion

Converting the datetime and hours & minutes into the day of the year and the time of the day in minutes.

[4]:
```
def convert_day(date):
    ''' Function for converting the date into the day of the year. '''

    year = str(date)[:4]
    month = str(date)[4:6]
    day = str(date)[6:]

    if year == "2012" or year == "2016":
        final = sum(month_1216[:int(month)-1])
```

```
        final = final + float(day)
    else:
        final = sum(month_len[:int(month)-1])
        final = final + float(day)
    return final

def convert_min(hour):
    ''' Function for converting the time into the time of the day in minutes. '''

    if len(str(int(hour)))==0: #if only one number for a minute (0 or 5)
        hours = 0
        minute = str(int(hour))[:]
    elif len(str(int(hour)))==1 or len(str(int(hour)))==2: #if only minutes
        hours = 0
        minute = str(int(hour))[:]

    elif len(str(int(hour)))==3: #if only 1 hour and minutes
        hours = str(int(hour))[:1]
        minute = str(int(hour))[1:]
    else: #if 2 numbers for hours and 2 numbers for minutes
        hours = str(int(hour))[:2]
        minute = str(int(hour))[2:]

    final = int(hours)*60+int(minute)
    return final
```

[5]:
```
month_1216 = [31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31] #leap year
month_len = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31] #normal year

day = data.date.apply(convert_day)
hour = data.time.apply(convert_min)
```

Converting the numbers into float or integer depending on the type of data.

[6]:
```
def str_to_float(str):
    ''' Function for converting string type into a float. '''

    try:
        return float(str)
    except:
        return float(str[0:4])

def str_to_int(str):
    ''' Function for converting the string type into an integer. '''

    try:
        return int(str)
```

```
    except:
        return int(str[0:4])
```

[7]:
```
#converting the values
data['day']= day
data['min'] = hour
data.temp = data.temp.apply(str_to_float)
data.date = pd.to_datetime(data.date, format = ('%Y%m%d'))
data.time = data.time.apply(str_to_int)

data.head()
```

[7]:
```
        date  time  temp  day  min
0 2011-01-01     5  -6.4  1.0    5
1 2011-01-01    10  -6.5  1.0   10
2 2011-01-01    15  -6.5  1.0   15
3 2011-01-01    20  -6.5  1.0   20
4 2011-01-01    25  -6.7  1.0   25
```

As you can see, all of the values are fine now. Now we will find the test/train sets. Since we are taking the full 2016 year as a test set, we will split our data like the following:

[8]:
```
split = np.where(data.date == '2016-01-01 00:00:00')[0][0]

X = data.iloc[:, 3:].values
y = data.iloc[:, 2].values

X_min = data.iloc[:, 4].values.reshape(-1,1)
X_day = data.iloc[:, 3].values.reshape(-1,1)
y_temp = data.iloc[:, 2].values.reshape(-1,1)

X_train_, X_test_ = X[:split], X[split:]
y_train_, y_test_ = y[:split].reshape(-1,1),  y[split:].reshape(-1,1)
```

Tasks

Instructions:

1. Cover each input dimension with a list of radial basis functions. This turns the pair of inputs into a much richer representation, mapping (d,t) into $(_1(d),_2(t))$. Experiment with different numbers of radial basis functions and different widths of the radial basis function in different dimensions.
2. Using this new representation, build a linear parameter model that captures both seasonal variations and daily variations.

Step 1: Linear Regression

Building a basic linear regression model, which will later be used in the main function `RBF()`.

```
[9]: def lin_reg(X_train, y_train, X_test, y_test, plot_days=False, plot_mins=False,␣
     ↪x_train=None, x_test=None, no_return=False):
         ''' Function for fitting the data into a Linear Regression and outputting
         results as the predicted values, R-squares scores and the MSE. '''

         #fitting a ridge regression and using it to make predictions
         regr = LinearRegression()
         regr.fit(X_train,y_train)

         y_pred_train = regr.predict(X_train)
         y_pred_test = regr.predict(X_test)

         #calculating the MSEs for both the training and testing sets
         mse_train = mean_squared_error(y_train, y_pred_train)
         mse_test = mean_squared_error(y_test, y_pred_test)

         #calculating the r-squared
         train_score = r2_score(y_train, y_pred_train)
         test_score = r2_score(y_test, y_pred_test)

         #outputting the plots when needed
         if plot_mins or plot_days:

             #the next 6 lines are needed so that there are no intersecting lines
             #when plotting the predicted values line
             train1 = pd.DataFrame()
             train1['x'], train1['y'], train1['y1']=x_train.reshape(1, -1)[0],␣
     ↪y_pred_train.reshape(1, -1)[0], y_train.reshape(1, -1)[0]

             test1 = pd.DataFrame()
             test1['x'], test1['y'], test1['y1']=x_test.reshape(1, -1)[0],␣
     ↪y_pred_test.reshape(1, -1)[0], y_test.reshape(1, -1)[0]

             train1 = train1.sort_values(by='x')
             test1 = test1.sort_values(by='x')

             #two plots: one for the training set, and the other for the test set
             fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 10))

             if plot_mins:
                 fig.suptitle('Contribution of minutes for train(left) and test␣
     ↪(right) sets')
                 ax1.set_xlabel('Minutes (train)')
                 ax2.set_xlabel('Minutes (test)')

             if plot_days:
```

```
            fig.suptitle('Contribution of days for train(left) and test (right)␣
↪sets')
            ax1.set_xlabel('Days (train)')
            ax2.set_xlabel('Days (test)')

        ax1.scatter(train1.x.values, train1.y1.values, label='True value')
        ax1.plot(train1.x.values, train1.y.values, label='Prediction', color='r')

        ax2.scatter(test1.x.values, test1.y1.values, label='True value')
        ax2.plot(test1.x.values, test1.y.values, label='Prediction', color='r')

        ax1.set_ylabel('Temperature (C)')
        ax2.set_ylabel('Temperature (C)')

        ax1.legend(loc='best')
        ax2.legend(loc='best')

    #output
    if not no_return:
        return mse_train, mse_test, train_score, test_score, y_pred_test
```

Step 2: RBF

Now, we will use the `lin_reg()` function we wrote above, as well as the `centers()` function below in the new function `RBF()`. It will output a table of results for a different number of radial basis functions and different widths.

[10]:
```
def centers(mini, maxi, n_rbf):
    ''' Function for creating n centers for the RBF.'''

    centers = []

    #base case for one center
    if n_rbf == 1:
        centers = [mini, maxi]

    #locating all the centers at equal intervals between min and max values
    else:
        centers = [i for i in range(mini, maxi, maxi//n_rbf)]

    #reshaping the list to fit the further operations - row to column
    centers = np.asarray((centers)).reshape(-1,1)
    return centers
```

[11]:
```
def RBF(n_rbf, widths):
    ''' Function for implementing the RBF functions. It will output a table
    with all results (i.e., MSE, R-squared).'''
```

```
test_results = []

for n in n_rbf:
    for width in widths:

        centers_min = centers(0, 1440, n) #centers for minutes
        centers_day = centers(0, 365, n) #centers for days

        #running the rbf from scikit
        rbf_min = rbf_kernel(X_min, centers_min, gamma=1/width)
        rbf_day = rbf_kernel(X_day, centers_day, gamma=1/width)

        #splitting the data after implementing rbf into train and test sets␣
↪for
        #minutes and days, individually
        X_train_min, X_test_min = rbf_min[:split], rbf_min[split:]
        X_train_day, X_test_day = rbf_day[:split], rbf_day[split:]

        #combining the data for minutes and days to get the data for the␣
↪full model
        X_train = np.concatenate((rbf_min[:split], rbf_day[:split]), axis=1)
        X_test = np.concatenate((rbf_min[split:], rbf_day[split:]), axis=1)

        #full model
        train_mse, test_mse, train_r2, r2, y_pred = lin_reg(X_train,␣
↪y_train_,
                                                            X_test, y_test_)

        #model for days
        train_mse_day, test_mse_day, train_r2_day, r2_day, y_pred_day =␣
↪lin_reg(X_train_day,
                                                                            ␣
↪   y_train_,
                                                                            ␣
↪   X_test_day,
                                                                            ␣
↪   y_test_)
        #model for minutes
        train_mse_min, test_mse_min, train_r2_min, r2_min, y_pred_min =␣
↪lin_reg(X_train_min,
                                                                            ␣
↪    y_train_,
                                                                            ␣
↪    X_test_min,
```

```
↪    y_test_)

            #storing all results
            test_results.append([n, width, r2, test_mse, train_r2, train_mse,␣
↪r2_day,
                                 test_mse_day, train_r2_day, train_mse_day,␣
↪r2_min, test_mse_min,
                                 train_r2_min, train_mse_min])


    #converting the results into a dataframe and showcasing the table of results
    test_results = pd.DataFrame(test_results, columns =
                                ["Number of centers", "Width","Full: Test R^2",␣
↪"Full: Test MSE","Full: Train R^2",
                                 "Full: Train MSE", "Day: Test R^2", "Days: Test␣
↪MSE",
                                 "Days: Train R^2", "Days: Train MSE", "Mins:␣
↪Test R^2",
                                 "Mins: Test MSE", "Mins: Train R^2", "Mins:␣
↪Train MSE"])

    return test_results
```

I ran the code on a different number of radial basis functions, as well as various widths. I combined the first two tasks from the assignment instructions by using the `RBF()` function. As you can see from the table below, we captured both the seasonal (i.e., day of the year) and daily (i.e., minute of the day) variations through our linear model.

Note: We could also change the width and number of the RBF list.

[12]:
```
rbf_res = RBF(n_rbf=[1, 10, 100, 180, 365], widths=[0.1, 1, 5, 10, 50])
rbf_res
```

[12]:

	Number of centers	Width	Full: Test R^2	Full: Test MSE	Full: Train R^2 \
0	1	0.1	0.005166	60.329094	0.005837
1	1	1.0	0.006280	60.261557	0.009409
2	1	5.0	0.010576	60.001048	0.016488
3	1	10.0	0.016530	59.639965	0.020786
4	1	50.0	0.053702	57.385732	0.037812
5	10	0.1	0.034288	58.563069	0.018620
6	10	1.0	0.074377	56.131966	0.043408
7	10	5.0	0.147820	51.678223	0.096260
8	10	10.0	0.188252	49.226303	0.134682
9	10	50.0	0.334551	40.354415	0.288595
10	100	0.1	0.151097	51.479500	0.192690
11	100	1.0	0.428717	34.643954	0.504298

12	100	5.0	0.550108	27.282517	0.633573
13	100	10.0	0.557038	26.862290	0.639483
14	100	50.0	0.571502	25.985134	0.657720
15	180	0.1	0.087390	55.342853	0.136780
16	180	1.0	0.532908	28.325589	0.624543
17	180	5.0	0.553732	27.062726	0.654548
18	180	10.0	0.559297	26.725266	0.660493
19	180	50.0	0.562146	26.552500	0.662763
20	365	0.1	0.558576	26.769029	0.664351
21	365	1.0	0.558996	26.743515	0.665078
22	365	5.0	0.558353	26.782506	0.665364
23	365	10.0	0.558189	26.792454	0.665360
24	365	50.0	0.556892	26.871149	0.662979

	Full: Train MSE	Day: Test R^2	Days: Test MSE	Days: Train R^2	\
0	60.900966	0.005007	60.338707	0.005662	
1	60.682138	0.005949	60.281621	0.009047	
2	60.248480	0.010242	60.021281	0.016124	
3	59.985204	0.016173	59.661629	0.020396	
4	58.942211	0.053099	57.422351	0.037159	
5	60.117896	0.033268	58.624911	0.017568	
6	58.599414	0.072816	56.226611	0.041861	
7	55.361741	0.145842	51.798173	0.094321	
8	53.008080	0.185783	49.376050	0.132252	
9	43.579593	0.329290	40.673434	0.283378	
10	49.454594	0.162082	50.813361	0.205623	
11	30.365980	0.412767	35.611191	0.488250	
12	22.446756	0.529460	28.534665	0.612744	
13	22.084712	0.531206	28.428751	0.613432	
14	20.967585	0.527759	28.637810	0.613605	
15	52.879546	0.262733	44.709600	0.308956	
16	22.999911	0.504981	30.019099	0.596428	
17	21.161850	0.517635	29.251741	0.618184	
18	20.797686	0.517380	29.267235	0.618252	
19	20.658627	0.517545	29.257214	0.618014	
20	20.561352	0.515121	29.404227	0.620753	
21	20.516794	0.515200	29.399444	0.620783	
22	20.499298	0.514703	29.429534	0.620897	
23	20.499530	0.514661	29.432130	0.620904	
24	20.645401	0.514093	29.466546	0.616408	

	Days: Train MSE	Mins: Test R^2	Mins: Test MSE	Mins: Train R^2	\
0	60.911640	0.000044	60.639694	0.000175	
1	60.704316	0.000216	60.629255	0.000363	
2	60.270790	0.000218	60.629128	0.000365	
3	60.009061	0.000242	60.627707	0.000390	
4	58.982187	0.000488	60.612761	0.000653	

5	60.182310	0.000905	60.587467	0.001052
6	58.694180	0.001446	60.554667	0.001548
7	55.480542	0.001864	60.529366	0.001940
8	53.156941	0.002356	60.499527	0.002430
9	43.899204	0.005149	60.330134	0.005216
10	48.662365	0.007190	60.206367	0.007357
11	31.349012	0.015869	59.680064	0.015995
12	23.722719	0.020579	59.394414	0.020719
13	23.680570	0.025759	59.080260	0.025916
14	23.670007	0.043671	57.994035	0.043891
15	42.332335	0.013442	59.827197	0.013702
16	24.722225	0.027817	58.955497	0.027968
17	23.389450	0.035996	58.459481	0.036182
18	23.385282	0.041819	58.106376	0.042030
19	23.399869	0.044281	57.957085	0.044512
20	23.232127	0.042398	58.071252	0.042665
21	23.230298	0.042556	58.061694	0.042811
22	23.223278	0.043368	58.012435	0.043509
23	23.222845	0.043504	58.004193	0.043610
24	23.498266	0.043406	58.010160	0.043518

Mins:	Train MSE
0	61.247789
1	61.236283
2	61.236144
3	61.234596
4	61.218482
5	61.194049
6	61.163704
7	61.139674
8	61.109629
9	60.939005
10	60.807808
11	60.278707
12	59.989271
13	59.670939
14	58.569806
15	60.419153
16	59.545212
17	59.042055
18	58.683807
19	58.531757
20	58.644919
21	58.635968
22	58.593217
23	58.587024
24	58.592658

As the number of RBF increases, there is a general trend that the average MSE decreases for all widths. In most cases, we could also see that an increase in the width leads to a lower MSE and a higher R^2 (this is not true for all cases, though). Overall, for our data, one of the best parameters is 365 functions with a width of 10 (i.e., lowest full test MSE - 20.499298). Another one is 100 functions with a width of 50, which gives us the highest test R-squared of 0.571502. We will choose 100/50 for our plots since it takes less time to execute, as well as the difference in MSEs is very small (around 0.5).

Although the train set and some of the test set results might be better for the higher number of RBF, this might lead to overfitting as the model would fit too closely on the training data. Thus, it won't be generalizable. To avoid that, we might want to choose a smaller number, such as 100/50 (functions/width).

Instructions:

3. Create two plots, one showing the time-of-day contribution, and one showing the time-of-year contribution.
4. (Optional) Make a 3D plot showing temperature as a function of (day, time). Make sure to label your axes!
5. Using R^2, quantify how your model performs on the testing data if you:
   - Train with just the daily component of the model
   - Train with just the yearly component of the model
   - Train with the full model.

Step 3: Plots

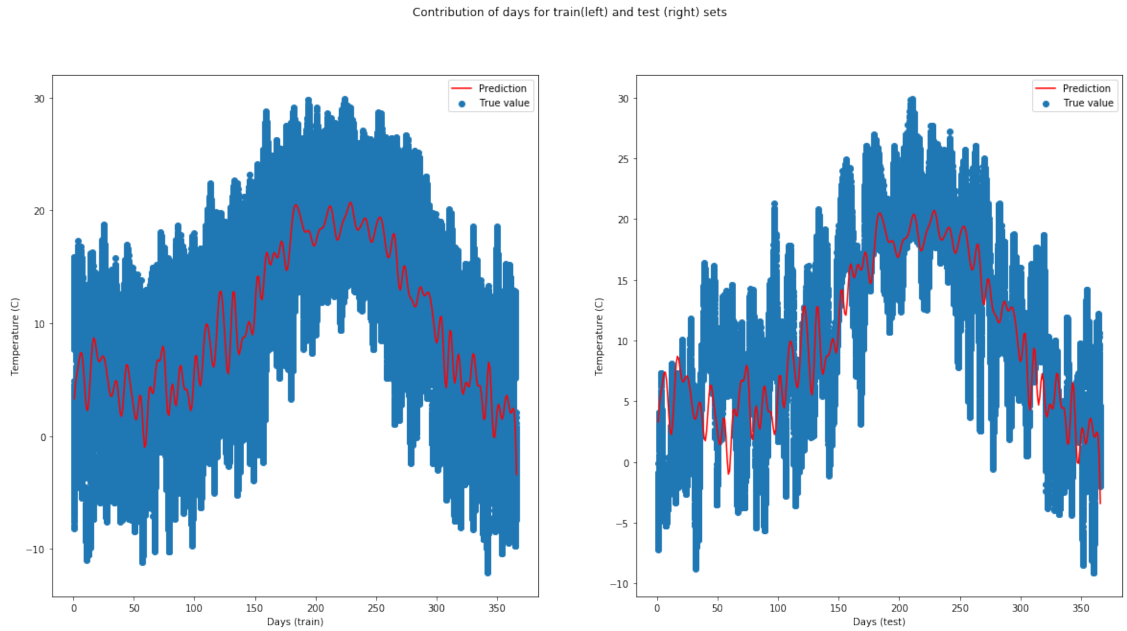
Below you can see the plots I have created for both contributions. I added the explanation for question 5 in some of the Step 3.

[13]:
```
n_day= 100
width_day = 50

#centers for days
center_day = centers(0, 365, n_day)

#finding rbf representations
rbf_train_day = rbf_kernel(X_day[:split], center_day, gamma = 1/width_day)
rbf_test_day = rbf_kernel(X_day[split:], center_day, gamma = 1/width_day)

#plotting
lin_reg(rbf_train_day, y_train_, rbf_test_day, y_test_, plot_days=True,
       x_train=X_day[:split], x_test=X_day[split:], no_return=True)
```



The above picture is the plot for the day-of-year contribution. We can see that the line we have obtained from the regression with RBF is pretty accurate, giving us the curve that fits the data. For the training data, it fits pretty well, while for the test data, it is still good, but there are some discrepancies in the values.

From the table in Step 2: RBF, we can see that the MSE and R^2 for the day-only model are very close to the full model's results (see table below).

Metric/Model	Full	Day
Test MSE	25.985	28.638
Train MSE	20.968	23.670
Test R^2	0.572	0.528
Train R^2	0.658	0.614

Still, the R-squared is only around 0.528, which is moderately good, meaning that 52.8% of the temperature variation could be explained through the variation in the day of the year.

The difference between the test set and train set error for both the individual day-of-year contribution and the full model is around 5.

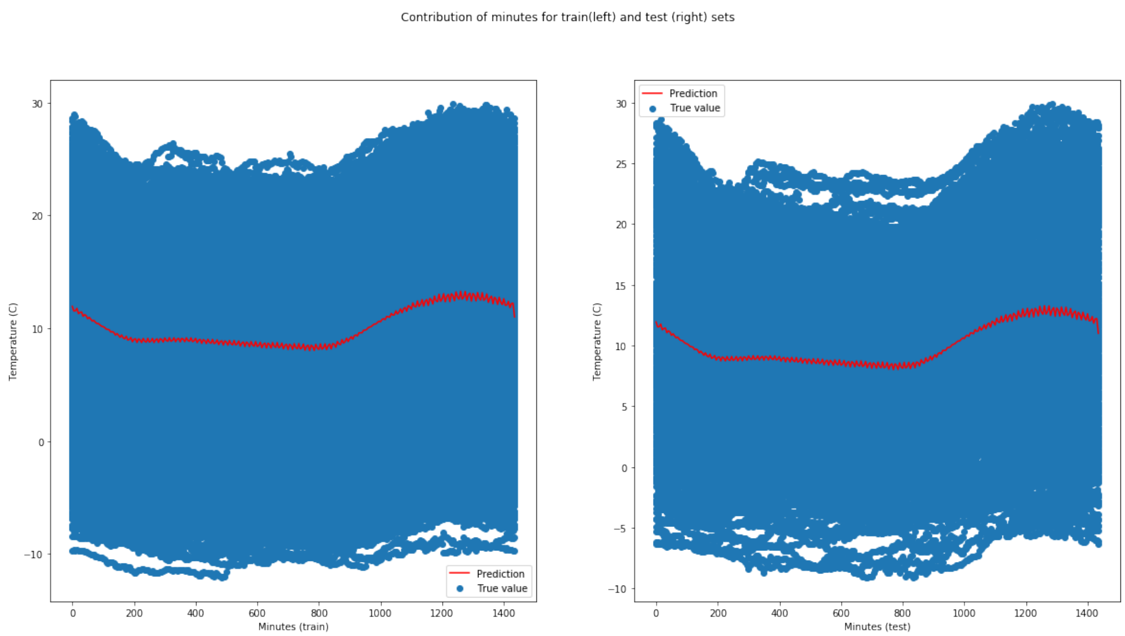
[14]:
```
n_min = 100
width_min = 50

#centers for days
center_min = centers(0, 1440, n_min)

#finding rbf representations
```

```
rbf_train_min = rbf_kernel(X_min[:split], center_min, gamma = 1/width_min)
rbf_test_min = rbf_kernel(X_min[split:], center_min, gamma = 1/width_min)

#plotting
lin_reg(rbf_train_min, y_train_, rbf_test_min, y_test_, plot_mins=True,
       x_train=X_min[:split], x_test=X_min[split:], no_return=True)
```



The above picture is the plot for the time-of-day contribution. We can see that the line we have obtained from the regression with RBF is almost flat, and the data itself has a large variance for both training and testing data. From the table in Step 2: RBF, we can see that the MSE and R^2 for the minute-only model are very far from the full model's results (see table below).

Metric/Model	Full	Minute
Test MSE	25.985	57.994
Train MSE	20.968	58.570
Test R^2	0.572	0.0437
Train R^2	0.658	0.0439

The test R-squared is only around 0.044, which is very small, meaning that only around 4.4% of the temperature variation could be explained through the variation in the minutes of the day. That makes sense because the time-of-day variation is not very representative of the general dataset. Our MSE is almost triple of the full model, which supports our previous point.

We can also observe that the total R-squared for the full model is equal to the sum of minute-only and day-only R-squared, which means that together, their variation explains around 57.2% of the variation in the temperature. It performs better than any individual model since it is more informed by containing both the minutes and days data.

Instructions:

Step 4: 3d Plot

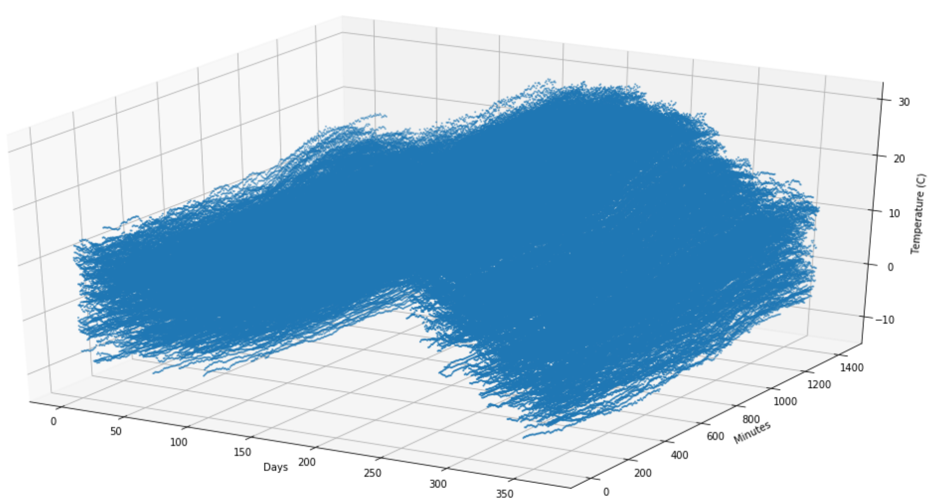
I plotted the 3d scatter for the temperature as a function of day and time. The plot represents all of the needed information. I didn't plot the predicted values, too, as it would have been too messy. We can see how it is super similar to the individual time-of-day or day-of-year contribution plots.

[15]:
```
from mpl_toolkits import mplot3d

#plotting the 3d figure
fig = plt.figure(figsize=(20, 10))
ax1 = fig.add_subplot(111, projection="3d")

#scatter
ax1.scatter(X_day, X_min, y_temp, s=0.8)
ax1.set_ylabel("Minutes")
ax1.set_zlabel("Temperature (C)")
ax1.set_xlabel("Days")
```

[15]:
```
Text(0.5, 0, 'Days')
```



Step 5: Report Continuation

We have already talked about this in Step 3, but we will contrast the RBF approach with the simple linear regression.

As we know, the original linear regression looks like a line through the data that minimizes the sum of squared residuals. Our data doesn't look linear at all, which is why the linear regression by itself does a terrible job of predicting the data. For example, the code snippet below shows that the MSE for normal linear regression is around 57, while the MSE for the RBF is below 30 for most cases with a high number of RBFs, as you could see from the table in Step 2: RBF). The R-squared for the RBF is almost ten times larger than for the basic linear regression (0.058 vs. 0.572). Thus, we can say that RBF performs much better for our data that couldn't easily be represented through a line.

[16]:
```
mse_tr, mse_te, r2_tr, r2_te, pred = lin_reg(X_train_, y_train_,
                                             X_test_, y_test_)


print("Training MSE:", mse_tr)
print("\nTesting MSE:", mse_te)
print("\nTraining R^2:", r2_tr)
print("\nTesting R^2:", r2_te)
```

```
Training MSE: 58.213173099191145

Testing MSE: 57.07497450381676

Training R^2: 0.04971282914792907

Testing R^2: 0.05882684466363486
```