# CS156 - Assignment 3

February 29, 2020

## 1 Yosemite Village Yearly Weather

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.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import csv

from sklearn import linear_model
from sklearn.metrics.pairwise import rbf_kernel
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score

import matplotlib.pyplot as plt
import matplotlib.cm as cm

import pandas as pd
```

### 1.1 Load data

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

#date, time, temp
usecols = [1, 2, 8]

#Load text in each file
data = [np.loadtxt(f, usecols=usecols) for f in files]
data = np.vstack(data)
print(data.shape)
```

```
(631296, 3)
```

### 1.1.1 Convert data into richer format

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.

[3]:
```
# Map from HHmm to an integer
data[:, 1] = np.floor_divide(data[:, 1], 100) * 60 + np.mod(data[:, 1], 100)
valid = data[:, 2] > -1000
```

[4]:
```
#check max for dates and minutes, looks good
print(max(data[:,0]))
print(max(data[:,1]))
```

```
20170101.0
1435.0
```

[5]:
```
#This is just times for now, we will add days next
X = data[valid, 1].reshape(-1, 1) #times of day
print(len(X))
Y = data[valid, 2] #temps
print(len(Y)) #same size, good
```

```
630854
630854
```

[6]:
```
#Now, I want to just take note of the location where we will split between
#training and test data, it will be used later after we have rbf transformed
#the data. This helps so I dont need to convert x_test and x_train separately
#at each step, I just work with all of the data, and split at the last step

location = np.where(np.logical_and(data[valid,0]== 20160101.0, data[valid,1]==0.
 ↪0))
location = location[0][0]

#For now, we use this location to create our test train split for the y
#values, we will use it for x later

y_train = Y[:location]
y_test = Y[location:]
```

[7]:
```
#Change our date to the day of the year. YYYYMMDD to #day out of 365 or 366
days = list(map((lambda i: datetime.datetime.strptime(str(i), '%Y%m%d.0').
 ↪timetuple().tm_yday), data[valid, 0]))
```

[8]:
```
#Now stack together our days of the year and times of the day
#because both of these are x's
#whereas our y that we will predict is temperature
#transpose so dimensions are correct
X = np.vstack((days, data[valid, 1])).T
print(X.shape) #days and times have been stacked
```

```
(630854, 2)
```

[9]:
```
#Check cols are still good
print(max(X[:,0]))
print(max(X[:,1]))
print(X[:,0].shape)
```

```
366.0
1435.0
(630854,)
```

## 1.2 Radial basis functions

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 ((d), (t)). Experiment with different numbers of radial basis functions and different widths of the radial basis function in different dimensions.

### 1.2.1 Parameter Selection

This section is to choose the best parameters for my model. The parameters that need to be decided upon are the - number of centers for day-of-the-year contribution - number of centers for time-of-the-day contribution - sigma for day-of-the-year contribution - sigma for time-of-the-day contribution - alpha

First, I build functions to engage in this process of parameter selection:

[39]:
```
'''
This function rbf transforms the X data and splits it into
test and training data for days of the year, time of the day,
and the full model of them together

'''

def rbf_transform_data(data, n_centers_days, n_centers_time,␣
 ↪sigma_days,sigma_time, split_location):

    #create centers, equally spaced
    centers_days = np.linspace(0, 366, n_centers_days).reshape(-1, 1) #366 days␣
 ↪a year
    centers_time = np.linspace(0, 1440, n_centers_time).reshape(-1, 1) #max of␣
 ↪mins
```

```
    #rbf transform
    days_rbf = rbf_kernel(data[:,0].reshape(-1,1),centers_days,gamma=1/
 ↪sigma_days)
    time_rbf = rbf_kernel(data[:,1].reshape(-1,1),centers_time,gamma=1/
 ↪sigma_time)

    #split into test and training
    train_days = days_rbf[:split_location]
    test_days = days_rbf[split_location:]

    train_time = time_rbf[:split_location]
    test_time = time_rbf[split_location:]

    #combined data for full model
    train_full = np.concatenate((train_days, train_time),axis=1)
    test_full = np.concatenate((test_days, test_time),axis=1)

    return train_days, test_days, train_time, test_time, train_full, test_full
```

[40]:
```
'''
Function to take in test and training data,
fit a ridge regression, and return mse's for
test and training data as well as r^2 values
'''

def test_one_model(x_train, y_train, x_test, y_test, alpha):

    #create and fit model
    regr = Ridge(alpha = alpha, fit_intercept=True) #ridge regression model
    regr.fit(x_train,y_train) #fitting ridge regression

    #predict for test and training
    y_train_pred = regr.predict(x_train)
    y_test_pred = regr.predict(x_test)

    #MSE for test and training sets
    mse_train = mean_squared_error(y_train, y_train_pred)
    mse_test = mean_squared_error(y_test, y_test_pred)

    #r^2 values
    r_train = regr.score(x_train, y_train)
    r_test = regr.score(x_test, y_test)

    return mse_train, mse_test, r_train, r_test
```

[41]:
```
'''
Function to get test and training MSE's for
time of day data, time of year data, and full model
'''

def test_three_models(data, n_centers_days, n_centers_time, sigma_days,␣
↪sigma_time, alpha, split_location, y_train, y_test):

    rbf_data = rbf_transform_data(data, n_centers_days, n_centers_time,␣
↪sigma_days, sigma_time, split_location)

    mse_train_full, mse_test_full = test_one_model(rbf_data[4], y_train,␣
↪rbf_data[5], y_test, alpha)[0:2]

    return n_centers_days, n_centers_time, sigma_days,sigma_time, alpha,␣
↪mse_train_full, mse_test_full
```

**Trying different number of rbfs and sigmas** In this section, I play around with different values for the number of rbs and the values for sigmas.

Because I had the constraint of runtime, I could not try too many values and was selection on the values I chose, which I explain below. For each combination of values, I present the MSE on the test and training data for the time-of-day, time-of-year, and full models. I use MSE because I want a metric that was sensitive to outliers in temperature to choose a model that best represents fluctuations in temperature.

In this section I try different numbers of rbfs for time-of-day and time-of-year. - For time-of-day I try 24 (hours in day), 60 (minutes in hour), 500 - For day-of-year I try 12 (because 12 months), 60, 200 - For sigmas I try various points in between the ranges of the values for days and times

Note, I do not use the maximum values as the number of centers because a) I want to avoid overfitting and b) it takes too long

Further work should definitely try more values, but this was a great start to get an idea of how paramaters behave in the model. Note, we keep alpha = 0.001 for this section and use cross validation to decide what value it should be later.

[42]:
```
centers_days = [12,60,200]
centers_time = [24,60,500]
sigmas_days = [10,100, 1000]
sigmas_time =  [1,10,100]

models = []
for i in sigmas_days:
    for j in sigmas_time:
        for k in centers_days:
            for p in centers_time:
                models.append(test_three_models(X, k, p, i,j, 0.001, location,␣
↪y_train, y_test))
```

[43]:
```
models
df = pd.DataFrame(models)
df.columns = ['# days centers', '# time centers','Sigma time','Sigma␣
 ↪days','Alpha','MSE Full train','MSE Full test']
```

[44]:
```
df.loc[:, df.columns != 'Alpha'].head()
```

[44]:
```
   # days centers  # time centers  Sigma time  Sigma days  MSE Full train  \
0              12              24          10           1       53.026370
1              12              60          10           1       52.671572
2              12             500          10           1       50.523186
3              60              24          10           1       25.133963
4              60              60          10           1       24.778829

   MSE Full test
0      51.489812
1      51.141835
2      49.030476
3      27.209279
4      26.861642
```

[45]:
```
#Show min training MSE
df[df['MSE Full train'] == df['MSE Full train'].min()]
```

[45]:
```
    # days centers  # time centers  Sigma time  Sigma days  Alpha  \
17             200             500          10          10  0.001

    MSE Full train  MSE Full test
17       20.606212      26.666455
```

**Selecting some parameters** We have found values for our paramters that leave us with the lowest MSE of all of the combinations we have tried. Note, we don't always choose the lowest MSE, because that can be overfitting, but since our MSE means we still deviate about 4.5 degrees, we are not worried about overfitting in this case. As I said before, using more centers may be more useful, but I did not have time with a x2 assignment as I already took an extension to play with more values so this should be done in an extension to this work!

The values we select are: - number of centers for day-of-the-year contribution = 200 - number of centers for time-of-the-day contribution = 500 - sigma for day-of-the-year contribution = 10 - sigma for time-of-the-day contribution = 10

[49]:
```
param_data = rbf_transform_data(X, 200, 500, 10,10, location)
```

**Selecting alpha using cross validation** For all of the experiments above, alpha stayed constant. In this section, we will change the value for alpha to figure out which is best.

Alpha is the factor that determines the regularisation strength which "improves the conditioning of the problem and reduces the variance of the estimates." In general, I have noticed a tighter fit with higher alphas. We explore very small alphas, to alphas of size 10

[48]:
```
#Alphas to test
alphas= [0.0001,0.001,0.01,0.1,1,3,10]
t = 0
store = []
for a in alphas:
    store.append(test_three_models(X, 200, 500, 100, 1000, a, location,␣
 ↪y_train, y_test))
    t+=1
    print(t)
```

```
1
2
3
4
5
6
7
```

[50]:
```
store
df2 = pd.DataFrame(store)
df2.columns = ['# days centers', '# time centers','Sigma time','Sigma␣
 ↪days','Alpha','MSE Full train','MSE Full test']
```

[51]:
```
df2.iloc[:,4:7]
```

[51]:

| | Alpha | MSE Full train | MSE Full test |
|---|---|---|---|
| 0 | 0.0001 | 21.434916 | 25.110663 |
| 1 | 0.0010 | 21.625735 | 24.881236 |
| 2 | 0.0100 | 21.826271 | 24.838631 |
| 3 | 0.1000 | 21.963643 | 24.916970 |
| 4 | 1.0000 | 22.084586 | 25.048534 |
| 5 | 3.0000 | 22.151934 | 25.065059 |
| 6 | 10.0000 | 22.233933 | 25.015906 |

[52]:
```
#find min
df[df['MSE Full train'] == df['MSE Full train'].min()]
```

[52]:

| | # days centers | # time centers | Sigma time | Sigma days | Alpha | \ |
|---|---|---|---|---|---|---|
| 17 | 200 | 500 | 10 | 10 | 0.001 | |

| | MSE Full train | MSE Full test |
|---|---|---|
| 17 | 20.606212 | 26.666455 |

[53]:
```
df[df['MSE Full test'] == df['MSE Full test'].min()]
```

[53]:

| | # days centers | # time centers | Sigma time | Sigma days | Alpha | \ |
|---|---|---|---|---|---|---|
| 74 | 12 | 500 | 1000 | 100 | 0.001 | |

| | MSE Full train | MSE Full test |
|---|---|---|

```
74        23.621661       23.177902
```

We get the best results with alpha = 0.001 so that is the value we will go with!

### 1.2.2 Build Linear parameter model using what we learned above

Using this new representation, build a linear parameter model that captures both seasonal variations and daily variations.

[54]:
```
#Parameters
alpha = 0.001
sigma_days = 10
sigma_time = 10
centers_days = 200
centers_time = 500
```

[55]:
```
final_data = rbf_transform_data(X, centers_days, centers_time,␣
 ↪sigma_days,sigma_time, location)
```

[56]:
```
print(final_data[4].shape)
```

```
(525479, 700)
```

[57]:
```
#create model and train
regr = Ridge(alpha = alpha, fit_intercept=True) #ridge regression model
regr.fit(final_data[4],y_train) #fitting ridge regression
```

[57]:
```
Ridge(alpha=0.001, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001)
```

[58]:
```
regr.predict(final_data[5])
regr.score(final_data[5],y_test)
```

[58]:
```
0.5602669665016669
```

### 1.2.3 Graphical representation of time-of-day and time-of-year contributions

**Time-of-day**

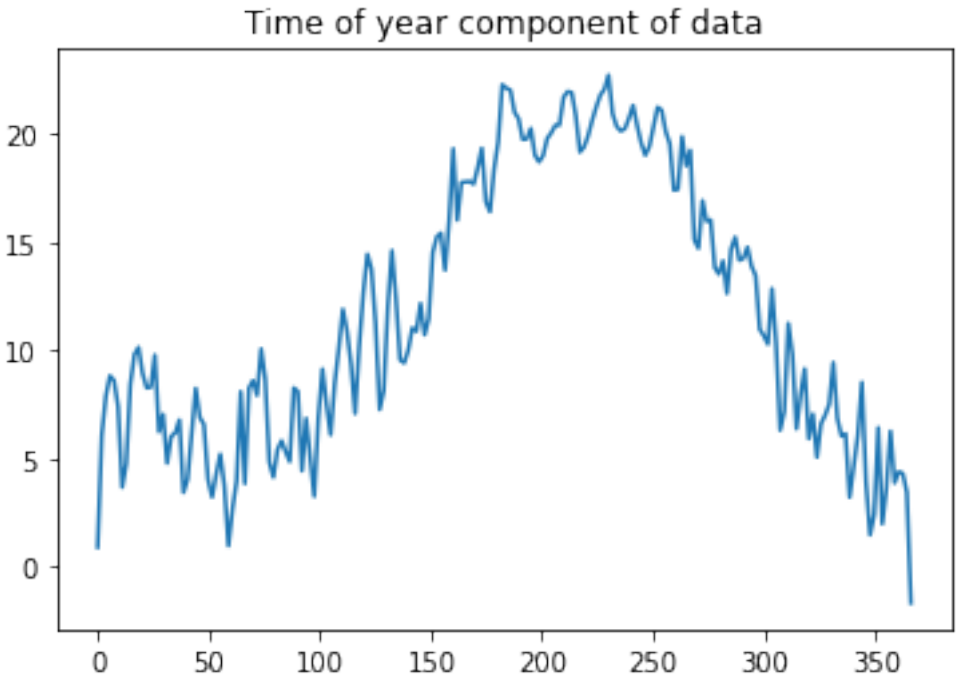
[59]:
```
centers_days_array = np.linspace(0, 366, centers_days).reshape(-1, 1) #366 days␣
 ↪a year
centers_time_array = np.linspace(0, 1440, centers_time).reshape(-1, 1) #max of␣
 ↪mins
```

[60]:
```
x_days = np.linspace(0,366,200)
x_time = np.zeros((200,)) #exclude time component

#transform data
rbf_days = rbf_kernel(x_days.reshape(-1,1),centers_days_array,gamma=1/
 ↪sigma_days)
rbf_time = rbf_kernel(x_time.reshape(-1,1),centers_time_array,gamma=1/
 ↪sigma_time)
```

```
print(rbf_days.shape,rbf_time.shape)
#combined data for full model
train_full = np.concatenate((rbf_days, rbf_time),axis=1)
print(train_full.shape)

y_days = regr.predict(train_full)
```

```
(200, 200) (200, 500)
(200, 700)
```

[61]:
```
#Plotting
plt.plot(x_days, y_days)
plt.title('Time of year component of data')
plt.show()
```



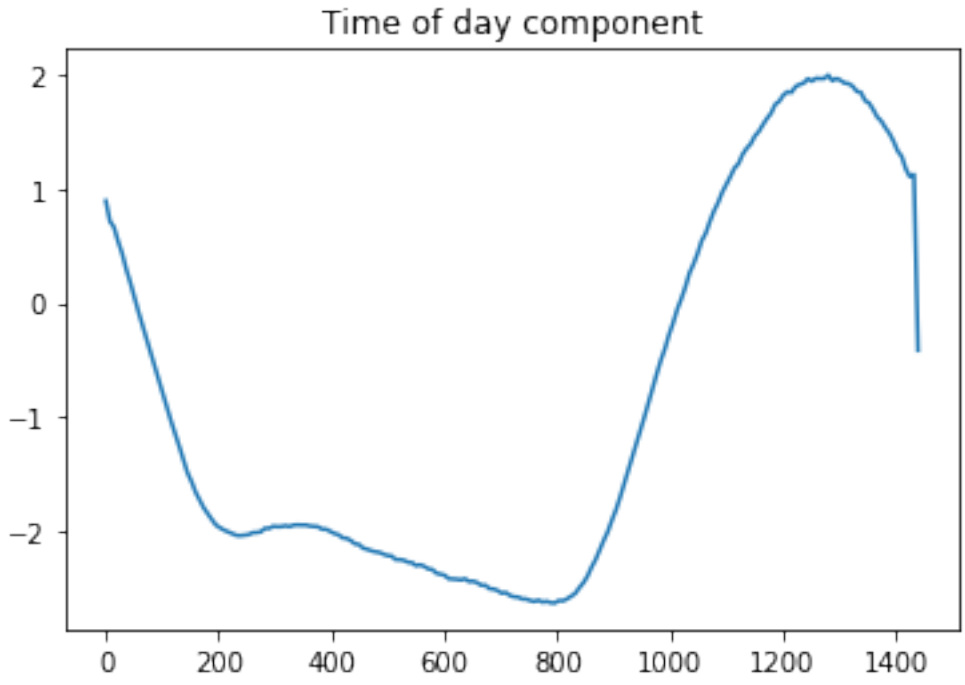

This plot makes sense since it shows hotter temperatures in summer months and colder winters.

[62]:
```
#Now with excluding day-of-year component
x_time = np.linspace(0,1440,200)
x_days= np.zeros((200,)) #exclude day component

#transform data
rbf_days = rbf_kernel(x_days.reshape(-1,1),centers_days_array,gamma=1/
 ↪sigma_days)
```

```
rbf_time = rbf_kernel(x_time.reshape(-1,1),centers_time_array,gamma=1/
 ↪sigma_time)
print(rbf_days.shape,rbf_time.shape)
#combined data for full model
train_full = np.concatenate((rbf_days, rbf_time),axis=1)
print(train_full.shape)

y_time = regr.predict(train_full)
```

```
(200, 200) (200, 500)
(200, 700)
```

[63]:
```
plt.plot(x_time, y_time)
plt.title('Time of day component')
plt.show()
```




This plot shows a distinctive hotter and cooler phase to each day, suggesting daytime and nighttime

**Daily component** These results show very low R^2 values if we only train using the time of day. This suggests that the variance in temperatures is not well explained by the time of day

[64]:
```
daily = test_one_model(final_data[2], y_train, final_data[3], y_test, alpha)
print('Training R^2:', daily[2])
print('Test R^2:', daily[3])
```

```
Training R^2: 0.04451233058598748
Test R^2: 0.0442806198322756
```

**Yearly component** These results show higher R^2 values (>0.5) if we only train using the day of year. This suggests that the variance in temperatures is more than 50% of the variance in the model is explained by the day of the year.

[65]:
```
yearly = test_one_model(final_data[0], y_train, final_data[1], y_test, alpha)
print('Training R^2:', yearly[2])
print('Test R^2:', yearly[3])
```

```
Training R^2: 0.6188864652989963
Test R^2: 0.5158879737814794
```

**Full** These results show even higher R^2 values (>0.6) if we train using the day of year and the time of day. This is because although time of day is not as informative than the yearly component, it is still helpful in predicting temperatures and improves the overall model so I would recommend using it in future models.

In general, we should note that the test and training R^2 values are similar, which means the model performs well on unseen data. We could definitely aim for higher R^2 values, but explaining more than 50% of the variance in the data is not too bad considering this is a small assignment and I did not run too many options for parameters.

[66]:
```
full = test_one_model(final_data[4], y_train, final_data[5], y_test, alpha)
print('Training R^2:', full[2])
print('Test R^2:', full[3])
```

```
Training R^2: 0.6636187641408666
Test R^2: 0.5602669665016669
```

### 1.2.4 HCs

#breakitdown: I thought a lot about how to break down the functions I needed to use in this assignment. Before, I had many smaller functions but it got really messy. I settled on 3 main functions so as not to get confused, and noticed that there are useful and unuseful ways to break down functions

#modeling: I spent time determining the best model parameters and was able to build a model that did a decent job at predicting temperatures