CS156 - Assignment 5 sparse gaussian processes

March 28, 2020

1 Yosemite Village Yearly Weather

In this assignment, I will use a Sparse Gaussian Process to estimate temperature at specific times of day and specific times of year. I will contrast my results to estimations using a linear parameter model from Assignment 3.

[86]:
```
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

import datetime

import GPy
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from GPy.models import SparseGPRegression
import time
import random
```

1.1 Load data

I load the data in a very similar way to Assignment 3, so citation credits to myself here.

[87]:
```
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.

[88]:
```
# Map from HHmm to an integer
data[:, 1] = np.floor_divide(data[:, 1], 100) * 60 + np.mod(data[:, 1], 100)
valid = data[:, 2] > -1000
```

[89]:
```
#check max for dates and minutes, looks good
print(max(data[:,0]))
print(max(data[:,1]))
```

```
20170101.0
1435.0
```

[90]:
```
#Now, I want to just take note of the location where we will split between
location = np.where(np.logical_and(data[valid,0]== 20160101.0, data[valid,1]==0.
 ↪0))
location = location[0][0]
```

[91]:
```
#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]))
```

[92]:
```
#create dataframe to store all data
df=pd.DataFrame()
df["days"]=days
df["minutes"]=data[valid,1]
df["temp"]=data[valid,2]
df['averages'] = 0
df['normalised'] = 0
df.shape
```

[92]:
```
(630854, 5)
```

[93]:
```
#check
df.head()
```

[93]:

	days	minutes	temp	averages	normalised
0	1	5.0	-6.4	0	0
1	1	10.0	-6.5	0	0
2	1	15.0	-6.5	0	0
3	1	20.0	-6.5	0	0
4	1	25.0	-6.7	0	0

[94]:
```
#split into training and test
data_train=df.iloc[:location,:]
data_test=df.iloc[location:,:]
```

[95]:
```
#randomly sample from the training and test data
#the full dataset was too large for my memory
train_samples = data_train.sample(10000, random_state = 123)
test_samples = data_test.sample(5000, random_state = 123)
```

[96]:
```
#we are computing average daily temperatures
#because the model can get confused with so much noise
averages = []
for i in range(1,367):
    temporary = df[df.days == i]
    averages.append(np.mean(temporary['temp']))
```

[97]:
```
#for each sample
for j in range(len(train_samples)):
    #get value of day
    day_val = train_samples.iloc[j,0]
    #get the average temp of that day
    av_temp = averages[day_val-1]

    #add this average to the relevant place in average column
    train_samples.iloc[j,3]=av_temp
    train_samples.iloc[j,4]=train_samples.iloc[j,2]-av_temp
```

[98]:
```
train_samples.head()
```

[98]:

	days	minutes	temp	averages	normalised
45027	157	650.0	0.0	12.533970	-12.533970
90018	314	110.0	6.4	6.483623	-0.083623
258703	169	115.0	15.7	15.303067	0.396933
194744	312	1385.0	14.7	9.722569	4.977431
243799	117	470.0	12.8	6.765741	6.034259

[101]:
```
#for each sample
for j in range(len(test_samples)):
    #get value of day
    day_val = test_samples.iloc[j,0]
    #get the average temp of that day
```

```
    av_temp = averages[day_val-1]

    #add this average to the relevant place in average column
    test_samples.iloc[j,3]=av_temp
    test_samples.iloc[j,4]=test_samples.iloc[j,2]-av_temp
```

[102]:
```
test_samples.head()
```

[102]:

	days	minutes	temp	averages	normalised
545713	71	370.0	5.9	5.184598	0.715402
609800	293	1130.0	14.5	12.362558	2.137442
543016	61	1285.0	12.7	2.521470	10.178530
586379	212	665.0	22.6	20.927836	1.672164
584256	205	130.0	21.7	19.178414	2.521586

[104]:
```
#separate into days, minutes, and temperature (y)
train_minutes=train_samples.minutes.values.reshape(-1,1)
normalised_temp_train =train_samples.normalised.values.reshape(-1,1)
average_temp_train =train_samples.averages.values.reshape(-1,1)
train_days=train_samples.days.values.reshape(-1,1)
y_train=train_samples.temp.values.reshape(-1,1)

test_minutes=test_samples.minutes.values.reshape(-1,1)
normalised_temp_test =test_samples.normalised.values.reshape(-1,1)
average_temp_test =test_samples.averages.values.reshape(-1,1)
test_days=test_samples.days.values.reshape(-1,1)
y_test=test_samples.temp.values.reshape(-1,1)
```

1.2 Sparse Gaussian Process

1.2.1 Time of Day Model

Here, I use a Sparse Gaussian Process to estimate temperature using the time-of-day component of the model.

Covariance function: Since this estimation is one dimensional, I chose a squared exponential kernel (Gaussian or rbf).

Hyperparameters: The hyperparameters (because I chose the squared exponential kernel) are the length scale and the variance. The length scale describes how close two points need to be to each other to influence eachother signicantly (Sterne, Session 8.1, 2020). In a squared exponential kernel a small length scale meansd that the function values will change quickly and large values will mean a smoother curve with slower changes. In this case, I chose 60 because based on Assignment 3, it seems reasonable that the previous 60 minutes influence the next estimation and also this is the value for 1 hour. I chose a variance of 0.5 which seems reasonable for temperatures measured in degrees celcius.

Sparse Gaussian Process: Sparse Gaussian Regression requires us to use inducing points. Inducing points can either be manually inputted or we can just give the number of inducing points we would like and GPy can place them optimally for us. I will use 60 inducing points in our x-range and allow GPy to place them optimally.

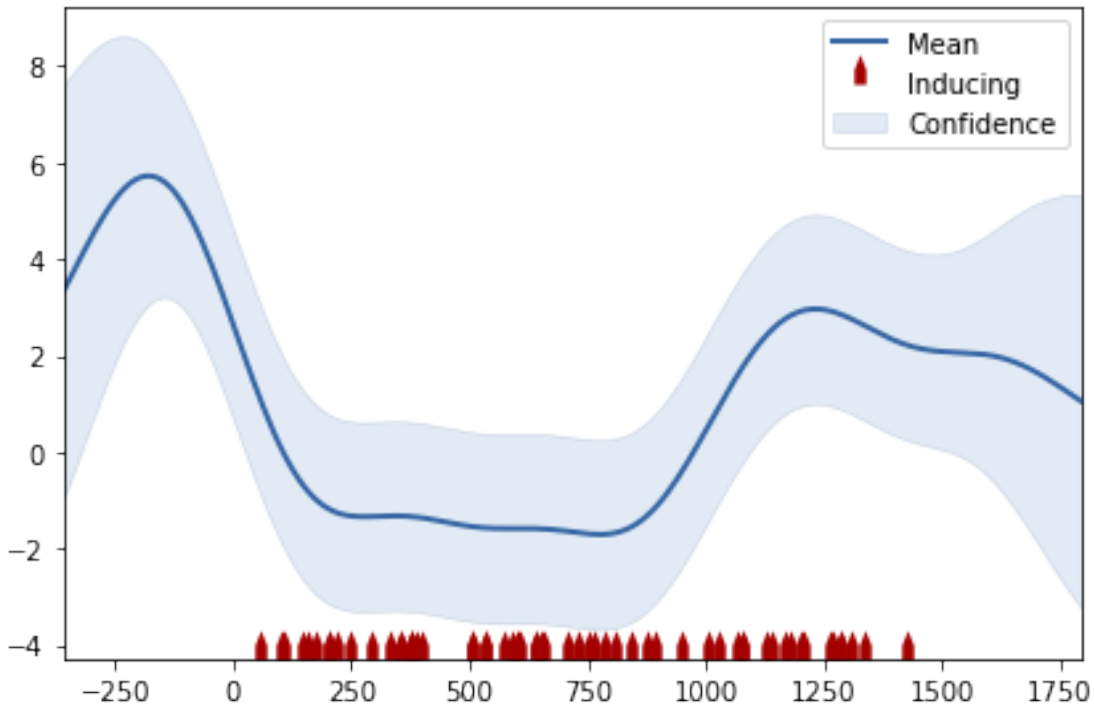
For this model, we use the normalised temperature where we have subtracted the daily average to try and remove random and seasonal variation.

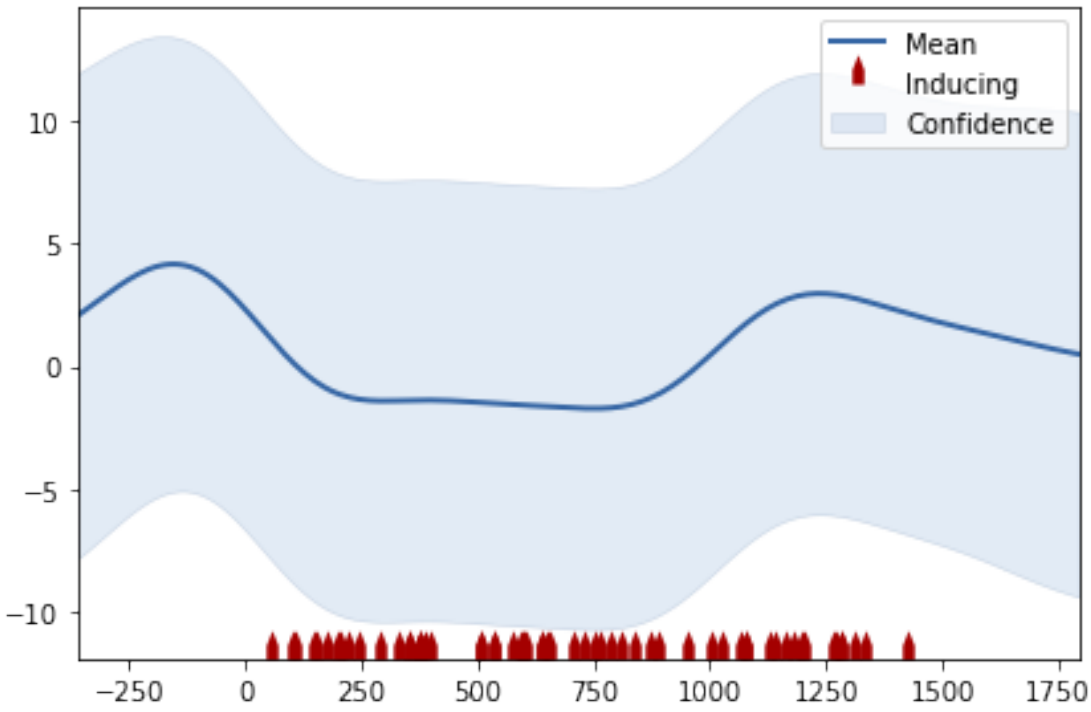
[109]:
```
np.random.seed(123)

#covariance function
k_mins = GPy.kern.RBF(1, lengthscale=60)

#Inducing clumps
Z = (np.linspace(0,1435,20))[:,None]
```

[114]:
```
#sparse GP:
t0=time.time()
m1 = GPy.models.SparseGPRegression(train_minutes,␣
 ↪normalised_temp_train,num_inducing = 60,kernel=k_mins)
t1=time.time()
m1.plot(plot_data=False)
print(m1)
m1.likelihood.variance = 0.5
print('Optimised:')
m1.optimize('bfgs')
t1=time.time()
m1.plot(plot_data=False)
print(m1)
total_time = t1-t0
```

```
Name : sparse_gp
Objective : 113945.5794046833
Number of Parameters : 63
Number of Optimization Parameters : 63
Updates : True
Parameters:
  sparse_gp.               |               value  |  constraints  |
priors
  inducing_inputs          |            (60, 1)  |               |
  rbf.variance             |   5.047318685431952  |      +ve      |
  rbf.lengthscale          |  215.47089871982956  |      +ve      |
  Gaussian_noise.variance  |                1.0  |      +ve      |
Optimised:

Name : sparse_gp
Objective : 29424.437788875508
Number of Parameters : 63
Number of Optimization Parameters : 63
Updates : True
Parameters:
  sparse_gp.               |               value  |  constraints  |
priors
```

```
inducing_inputs          |              (60, 1)  |              |
rbf.variance             |   5.056545816368159   |      +ve     |
rbf.lengthscale          |  215.47005342957618   |      +ve     |
Gaussian_noise.variance  |  20.966165369383894   |      +ve     |
```





[ ]:
```
#function to compute MSE
def mse(y,pred):
    return np.mean([(y[i]-pred[0][i])**2 for i in range(len(y))])
```

[138]:
```
print("The training MSE is",mse(normalised_temp_train,m1.
 ↪predict(train_minutes)))
print("The test MSE is",(mse(normalised_temp_test,m1.predict(test_minutes))))
```

```
The training MSE is 20.947059648345363
The test MSE is 19.296030394173933
```

[123]:
```
print("The model took:",total_time, 'seconds to run')
```

```
The model took: 35.994539976119995 seconds to run
```

We see that we have a rbf variance and lengthscale of the same amounts we specified which good according to this tuturial: https://nbviewer.jupyter.org/github/SheffieldML/notebook/blob/master/GPy/sparse_gp_regression.ipynb

But, I will try

1.2.2 Time of Year Model

Here, I use a Sparse Gaussian Process to estimate temperature using the time-of-year component of the model.

Covariance function: Since this estimation is one dimensional, I also chose a squared exponential kernel (Gaussian or rbf).

Hyperparameters: In this case, I chose 30 because based on Assignment 3, it seems reasonable that the previous 30 minutes influence the next estimation, there are also 30 days in a month so this seemed reasonable. I chose a likelihood variance of 0.5 which seems reasonable for temperatures measured in degrees celcius.

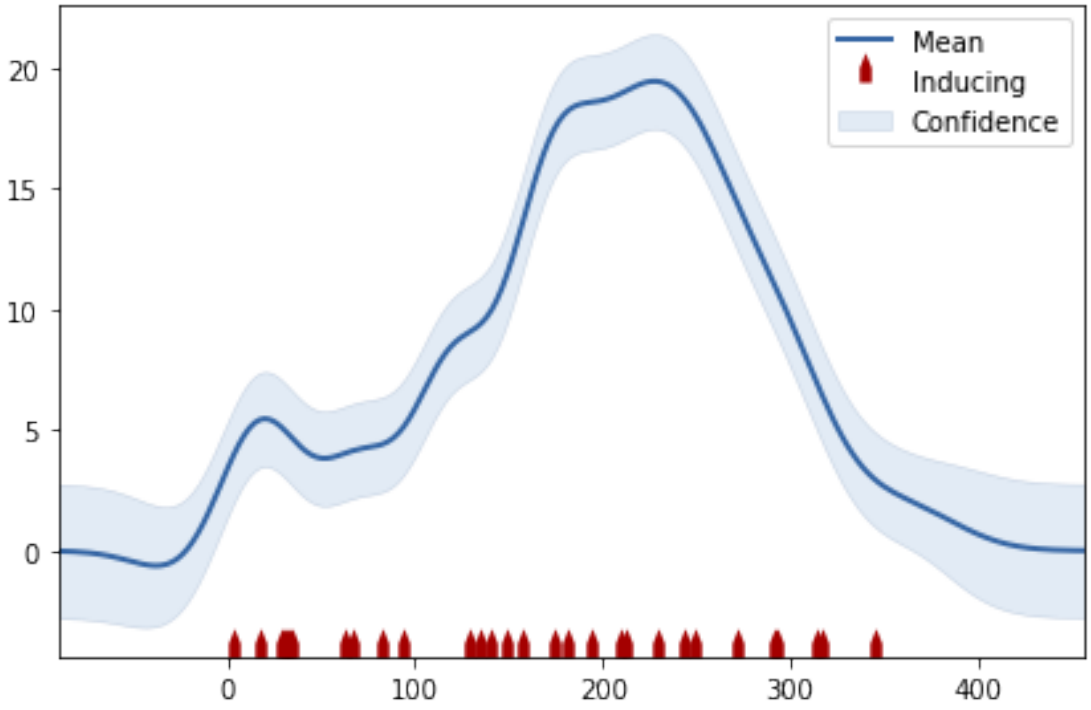
Sparse Gaussian Process: I will use 30 equally spaced inducing points in our x-range. It seems reasonble for 366 days to have 30 important points for changes.

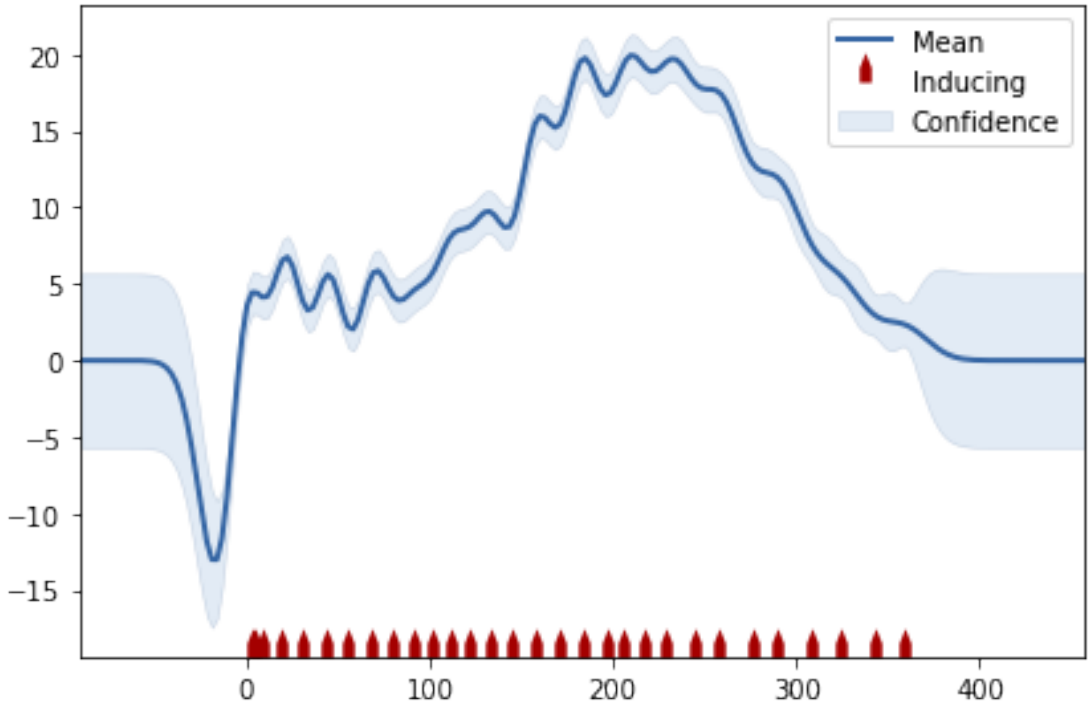
For this model I use the daily average temperatures.

[115]:
```
k_days = GPy.kern.RBF(1, lengthscale=30.)
```

[116]:
```
t0=time.time()
m2 = GPy.models.SparseGPRegression(train_days, average_temp_train, num_inducing␣
 ↪= 30,kernel=k_days)
m2.plot(plot_data=False)
print(m2)
m2.optimize('bfgs')
t1=time.time()
m2.likelihood.variance = 0.5
m2.plot(plot_data=False)
print(m2)
total_time2 = t1-t0
```

```
Name : sparse_gp
Objective : 21595.576847719843
Number of Parameters : 33
Number of Optimization Parameters : 33
Updates : True
Parameters:
  sparse_gp.               |    value  |  constraints  |  priors
  inducing_inputs          |  (30, 1)  |               |
  rbf.variance             |      1.0  |      +ve      |
  rbf.lengthscale          |     30.0  |      +ve      |
  Gaussian_noise.variance  |      1.0  |      +ve      |

Name : sparse_gp
Objective : 24552.797125806333
Number of Parameters : 33
Number of Optimization Parameters : 33
Updates : True
Parameters:
  sparse_gp.               |               value  |  constraints  |
priors
  inducing_inputs          |             (30, 1)  |               |
  rbf.variance             |   7.939799132655016  |      +ve      |
  rbf.lengthscale          |  13.072886830767398  |      +ve      |
  Gaussian_noise.variance  |                 0.5  |      +ve      |
```


Mean
Inducing
Confidence
20
15
10
5
0
0
100
200
300
400


Mean
Inducing
Confidence
20
15
10
5
0
−5
−10
−15
0
100
200
300
400

[139]:
```
print("The training MSE is",mse(average_temp_train,m2.predict(train_days)))
print("The test MSE is",(mse(average_temp_test,m2.predict(test_days))))
print("The model took:",total_time2, 'seconds to run')
```

```
The training MSE is 1.7773128788190635
The test MSE is 1.760048699109984
The model took: 737.6568231582642 seconds to run
```

1.2.3 Full Model

We can do a 2 dimensional sparse gaussian process in this case because we have time of day as well as time of year. This allows interaction between both components and allows us to visualise this.

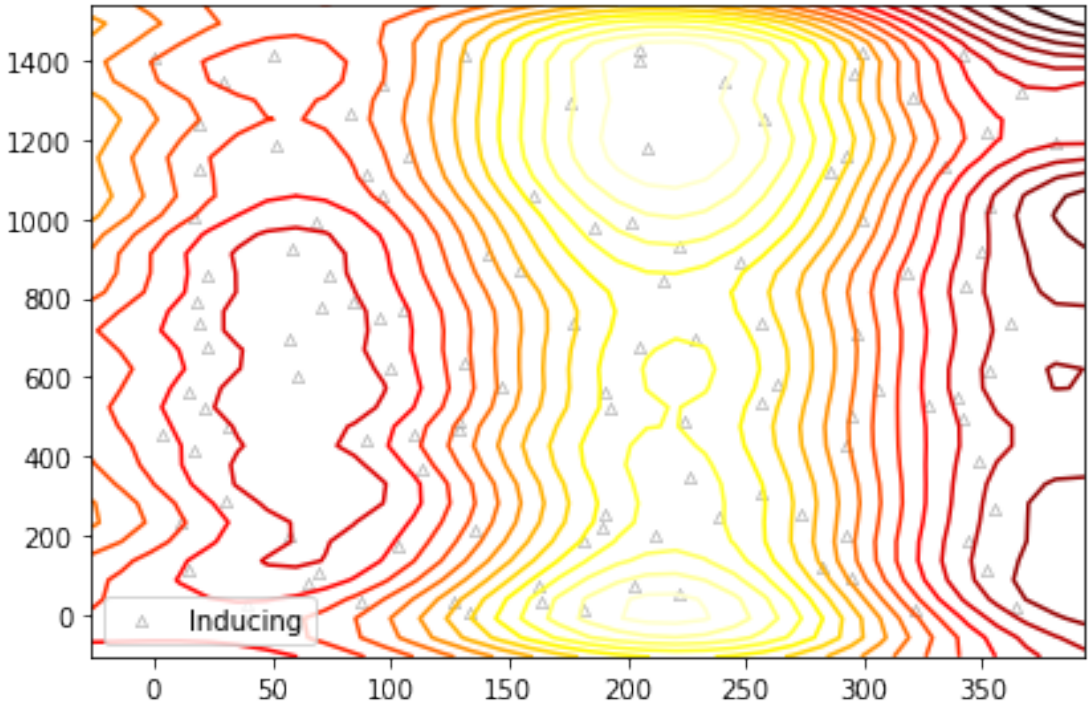
[35]:
```
#stack days and minutes data together because we will use both
X_train = np.concatenate((train_days, train_minutes),axis=1)
X_test = np.concatenate((test_days, test_minutes),axis=1)
```

Now that we have a 2 dimensional model, we need a 2 dimensional covariance function. I choose the product of 2 squared exponentials (Gaussian or rbf) for similar reasons to above.

[36]:
```
k3 = GPy.kern.RBF(2, lengthscale=100.)
```

[121]:
```
t0=time.time()
m3 = GPy.models.SparseGPRegression(X_train, y_train,num_inducing =␣
 ↪120,kernel=k3)
m3.likelihood.variance = 0.5
m3.optimize('bfgs')
t1=time.time()
m3.plot(plot_data=False)
print(m3)
total_time3 = t1-t0
```

```
Name : sparse_gp
Objective : 30061.257784100828
Number of Parameters : 243
Number of Optimization Parameters : 243
Updates : True
Parameters:
  sparse_gp.                |                value  |  constraints  |
priors
  inducing_inputs           |             (120, 2)  |               |
  rbf.variance              |  119.21954220867867   |      +ve      |
  rbf.lengthscale           |   121.4185560119071   |      +ve      |
  Gaussian_noise.variance   |   23.11075183824341   |      +ve      |
```



In the plot above, yellow represents higher numbers and darker colors represent lower numbers. The day of the year is on the x axis and the time of day is on the y axis. This shows a warmer trend around period 200 which is consistent with our results in assignment 3 (the summer) and a cooler time in the winter at the beginning and end of the year. This suggests our data is in the northern hempishere, which Yosemite is. Time of day variations are more difficult to see, but we would expect to see more heat at time 1200 compared to 800 because if our results were consistent with the LPR in assignment 3. We see that if we follow the link up from x=150, the color is darker at y=800 than it is at y=1200 suggesting that it is hotter at y=1200.

[141]:
```
print("The training MSE is",mse(y_train,m3.predict(X_train)))
print("The test MSE is",(mse(y_test,m3.predict(X_test))))
print("The model took:",total_time3, 'seconds to run')
```

```
The training MSE is 22.940590223297495
The test MSE is 22.52846523980916
The model took: 396.4613480567932 seconds to run
```

1.2.4 Linear Parameter restults from assignment 3

In assignment 3, I computed an MSE of 23.34413 for the training set and 26.666455 for the test set for the full model. The model took 12.96 seconds to run.

1.3 Discussion

In this section, I will compare the MSE restults and training time results from the linear parameter model from assignment 3 and then discuss these differences.

We can see that the MSE for both the training and test set using the Sparse Gaussian Process is better than the linear parameter model. We also see very similar MSE's between the GP between the test and training set which tells us that the model is performing well on unseen data. The GP took much longer to run because it is a more complicated process. This is possibly because in the LPR we manually choose the centers, and here the optimise method needs to find the best inducing points for our model amongst other things that make it longer.

I think I could get even better results if I had more practice at choosing covariance functions.

1.4 HCs

#descriptivestats: I compute and analyse the MSE for the LPR and GP and compare the results, justifying which is more appropriate to sue

#constraints: I realised that memory was a constraint to solving this problem and randomly sampled subsets of the data to adhere to this constraint and be able to run my model

#multiplecauses: I identify how day of the year and time of the day interact to produce a temperature at that specific point in time. By including both aspects of my model and explaining that they both contribute, I demonstrate an understanding of how systems are not a simple input and output but rather an interaction of causes to produce complex results.