CSE 569 Fundamental of Statistical Learning HW2 Solutions

Solution1

Part (a)

The maximum likelihood estimate is given by the product of IID probabilities for number of meteorites hitting the moon surface

$$p(D|\lambda) = \prod_{i=1}^{n} p(x_i|\lambda)$$

$$p(D|\lambda) = \prod_{i=1}^{n} \frac{\lambda^{x_i} e^{-\lambda}}{x_i!}$$

$$p(D|\lambda) = e^{-n\lambda} \prod_{i=1}^{n} \frac{\lambda^{x_i}}{x_i!}$$

Representing in log likelihood form we get,

$$\ln(p(D|\lambda)) = \ln(e^{-n\lambda}) + \sum_{i=1}^{n} \ln(\frac{\lambda^{x_i}}{x_i!})$$

To find the maximum estimate for λ , we differentiate $\frac{\partial L}{\partial \lambda}$ w.r.t 0

$$0 = -n + \sum_{i=1}^{n} \frac{x_i}{\lambda}$$

$$\lambda = \frac{1}{n} \sum_{i=1}^{n} x_i$$

The given observations are {45,36,15,25,40}

Putting this values in the above equation we get λ as

$$\lambda = \frac{45 + 36 + 15 + 25 + 40}{5}$$

$$\lambda = 32.2$$

Part (b)

Using the fact that $p(\lambda)$ is a distribution and thus integrate to 1 over the space of λ

$$\int_0^\infty \frac{\beta^\alpha \lambda^{\alpha-1} \exp(-\beta\lambda) d\lambda}{\Gamma(\alpha)} = 1$$

$$\int_0^\infty \lambda^{\alpha-1} \exp(-\beta\lambda) d\lambda = \frac{\Gamma(\alpha)}{\beta^\alpha}$$

Part (c)

By using Bayes estimation , we estimate the posterior distribution as follows

$$p(\lambda \mid D) = \frac{p(D \mid \lambda) p(\lambda)}{p(D)}$$

We can write the likelihood expression as

$$p(D \mid \lambda) = e^{-n\lambda} \prod_{i=1}^{n} \frac{1}{x_i!} \cdot \prod_{i=1}^{n} \lambda^{x_i}$$

By considering k as the constant we can write the above equation as follows

$$p(D \mid \lambda) = k e^{-n\lambda} . \lambda^{\sum_{i=1}^n x_i} \text{ ,where } k = \prod_{i=1}^{n} \frac{1}{x_i!}$$

Using Bayes formulation, we can write the posterior distribution as

$$p(\lambda \mid D) = \frac{p(D \mid \lambda) p(\lambda)}{\int_0^\infty p(D \mid \lambda) p(\lambda) d\lambda}$$

$$p(\lambda \mid D) = \frac{k e^{-n\lambda} . \lambda^{\sum_{i=1}^n x_i} p(\lambda)}{\int_0^\infty k e^{-n\lambda} . \lambda^{\sum_{i=1}^n x_i} p(\lambda) d\lambda}$$

$$p(\lambda|D) = \frac{ke^{-n\lambda}\,.\,\lambda^{\sum_{i=1}^{n} x_i}\frac{\beta^{\alpha}\lambda^{\alpha-1}\exp(-\beta\lambda)}{\Gamma(\alpha)}}{\int_0^{\infty} ke^{-n\lambda}\,.\,\lambda^{\sum_{i=1}^{n} x_i}\frac{\beta^{\alpha}\lambda^{\alpha-1}\exp(-\beta\lambda)}{\Gamma(\alpha)}d\lambda}$$

We cancel out all the constant (i.e non λ terms)

$$p(\lambda|D) = \frac{\exp(-n\lambda)\,.\,\lambda^{\sum_{i=1}^{n} x_i+\alpha-1}\exp(-\beta\lambda)}{\int_0^{\infty}\exp(-n\lambda)\,.\,\lambda^{\sum_{i=1}^{n} x_i+\alpha-1}\exp(-\beta\lambda)d\lambda}$$

$$p(\lambda|D) = \frac{\exp(-(n+\beta)\lambda)\,.\,\lambda^{\sum_{i=1}^{n} x_i+\alpha-1}}{\int_0^{\infty}\exp(-(n+\beta)\lambda)\,.\,\lambda^{\sum_{i=1}^{n} x_i+\alpha-1}d\lambda}$$

We know from Part b derivation,

$$\int_0^{\infty}\lambda^{\alpha-1}\exp(-\beta\lambda)d\lambda = \frac{\Gamma(\alpha)}{\beta^{\alpha}}$$

Using the above equation we get,

$$p(\lambda|D) = \frac{\exp(-(n+\beta)\lambda)\,.\,\lambda^{\sum_{i=1}^{n} x_i+\alpha-1}(\beta+n)^{\alpha+\sum_{i=1}^{n} x_i}}{\Gamma(\sum_{i=1}^{n} x_i+\alpha)}$$

We can see that above posterior expression is the conjugate prior of the gamma distribution $p(\lambda)$ as it is represented in the similar form with parameters as follows

$$\alpha^* = \alpha + \sum_{i=1}^{n} x_i \text{ and } \beta^* = \beta + n$$

Solution 2

Code for PCA

```
import matplotlib.pyplot as plt
import numpy as np
from numpy import linalg as la

import sklearn
from sklearn.decomposition import PCA
from sklearn.decomposition import TruncatedSVD
from sklearn.preprocessing import StandardScaler

%run load_mnist.py

# Preparing the data
trX, trY, tsX, tsY = mnist(noTrSamples=400,
                               noTsSamples=100, digit_range=[5,
8],noTrPerClass=200, noTsPerClass=50)
trX = trX.T
trY = trY.T

tsX = tsX.T
tsY = tsY.T

trX_class_5 = trX[np.where(trY == 5)[0]]
trX_class_8 = trX[np.where(trY == 8)[0]]
tsX_class_5 = tsX[np.where(tsY == 5)[0]]
tsX_class_8 = tsX[np.where(tsY == 8)[0]]


def run_PCA(inp,k=10):
    pca = PCA(n_components=int(k))
    object_concept_matrix = pca.fit_transform(inp)
    Vt = pca.components_
    return object_concept_matrix,Vt


def plot_images(data,title):
    f = plt.figure()
    fig, ax = plt.subplots(1, 5)
    fig.suptitle('Images for %s'%title)
    for i in range(0,5):
        ax[i].imshow((data[i,:]).reshape(28,28))
    plt.show()
```

```
#PCA on train data
pca_transf_data,Vtrain = run_PCA(trX)
#print(pca_transf_data.shape)

pca_transf_testdata,Vtest = run_PCA(tsX)

#Plot covariance matrix
cov_pca = pca_transf_data.T @ pca_transf_data #shape k x k (10x10)
plt.matshow(cov_pca)

#Plot reconstructed images and original images
reconstructed_data = pca_transf_data @ Vtrain

reconstructed_data_class_5 = reconstructed_data[np.where(trY == 5)
[0]]
reconstructed_data_class_8 = reconstructed_data[np.where(trY == 8)
[0]]

#Plotting digit 5 transformed images

plot_images(trX_class_5,'Original Images')
plot_images(reconstructed_data_class_5,'PCA Reconstructed Images')

plot_images(trX_class_8,'Original Images')
plot_images(reconstructed_data_class_8,'PCA Reconstructed Images')
```

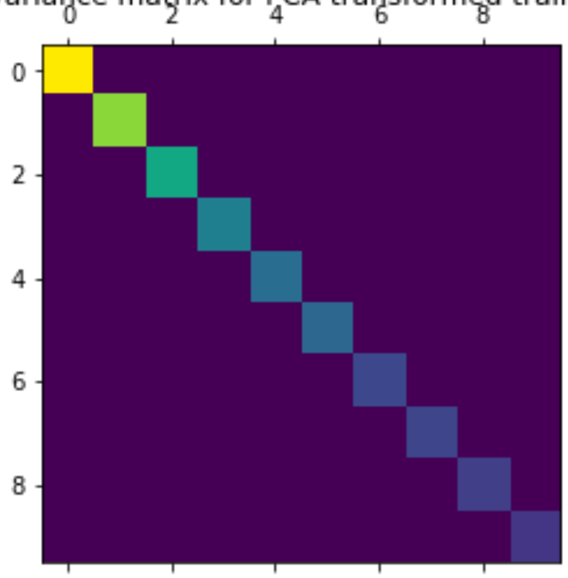
Code for LDA

```
def get_predictions(w,X,threshold):
    projected_data = np.dot(X,w)
    predicted_labels = np.select([projected_data <= threshold,
projected_data>threshold], [np.zeros_like(projected_data),
np.ones_like(projected_data)])
    predicted_labels = np.array([5 if i[0] == 0 else 8 for i in
predicted_labels.tolist()])
    return predicted_labels

pca_transf_trdata_class_5 = pca_transf_data[np.where(trY == 5)[0]]
pca_transf_trdata_class_8 = pca_transf_data[np.where(trY == 8)[0]]

pca_transf_testdata_class_5 = pca_transf_testdata[np.where(tsY ==
5)[0]]
pca_transf_testdata_class_8 = pca_transf_testdata[np.where(tsY ==
8)[0]]

#mean of all the points of class 5
m1_bar = np.mean(pca_transf_trdata_class_5,axis=0).reshape(-1,1)

#mean of all the points of class 8
m2_bar = np.mean(pca_transf_trdata_class_8,axis=0).reshape(-1,1)
```

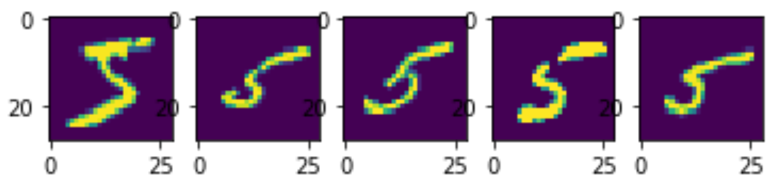
```
#number of class 5 images
n1 = int((trY == 5).sum())

#number of class 8 images
n2 = int((trY == 8).sum())

#Sw1 square matrix for class5 data
Sw1 = (1/n1)*np.dot(pca_transf_trdata_class_5.T - m1_bar,
(pca_transf_trdata_class_5.T - m1_bar).T)

#Sw2 square matrix for class8 data
Sw2 = (1/n2)*np.dot(pca_transf_trdata_class_8.T - m2_bar,
(pca_transf_trdata_class_8.T - m2_bar).T)

Sw = Sw1 +  Sw2

#computing the direction of LDA in new projected space
w = np.linalg.pinv(Sw) @ (m2_bar - m1_bar)

m1 = np.dot(w.T,m1_bar)
m2 = np.dot(w.T,m2_bar)

m1 = m1[0][0]
m2 = m2[0][0]

threshold = (m1 + m2) / 2

print("Threshold for LDA",threshold)

training_pred = get_predictions(w,pca_transf_data,threshold)
training_accuracy = (training_pred == trY.flatten()).mean() * 100
print("Training_Accuracy",training_accuracy)

test_pred = get_predictions(w,pca_transf_testdata,threshold)
testing_accuracy = (test_pred == tsY.flatten()).mean() * 100
print("Testing_Accuracy",testing_accuracy)
```

Outputs

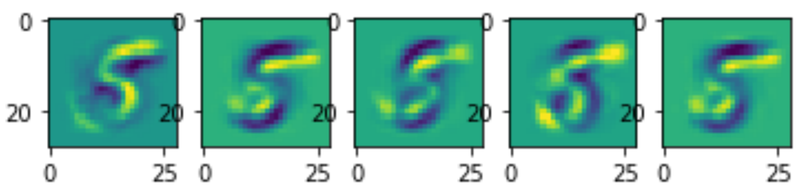
Covariance matrix for PCA transformed train data



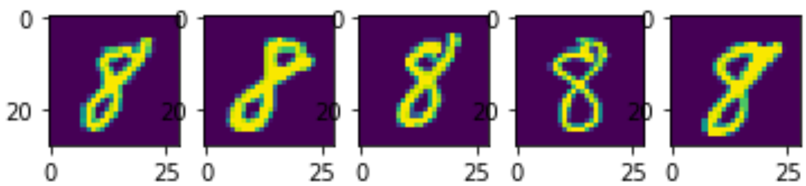
Images for Original Images for Digit 5



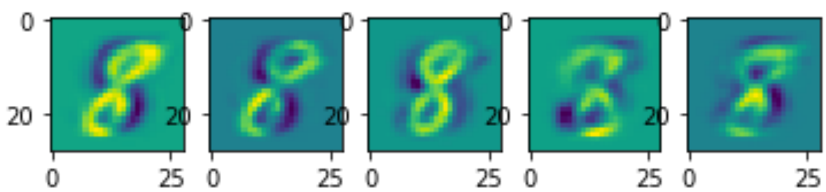
Images for PCA Reconstructed Images for Digit 5



Images for Original Images for Digit 8



Images for PCA Reconstructed Images for Digit 8



Solution 3

Code for declarations

```
import numpy as np

pi_1 = np.array([0.0,1.0,0.0,0.0])
a_1 = np.array([[1.0,0.0,0.0,0.0],
                [0.0,0.0,0.0,1.0],
                [0.0,0.4,0.3,0.3],
                [0.3,0.2,0.2,0.3]
               ])
b_1 = np.array([[1.0,0.0,0.0,0.0,0.0],
                [0.0,0.5,0.5,0.0,0.0],
                [0.0,0.2,0.2,0.3,0.3],
                [0.0,0.0,0.0,0.5,0.5]
               ])

pi_2 = np.array([0.0,0.0,0.0,1.0])
a_2 = np.array([[1.0,0.0,0.0,0.0],
                [0.1,0.3,0.5,0.1],
                [0.1,0.4,0.3,0.2],
                [0.1,0.4,0.2,0.3]
               ])
b_2 = np.array([[1.0,0.0,0.0,0.0,0.0],
                [0.0,0.0,0.5,0.0,0.5],
                [0.0,0.0,0.5,0.5,0.0],
                [0.0,0.5,0.0,0.0,0.5]
               ])

sequences_map = {'S':0,'A':1,'B':2,'C':3,'D':4}

reverse_sequences_map = dict((v,k) for k,v in
sequences_map.items())

states = [0,1,2,3]
```

Part a)

Code for generating 10 sequences of observations from HMM with λ_1

```
def gen_sequences(b,a):
    gen_sequences = []
    for _ in range(10):
        sequences = []
        q = 1
        while True:
            bq = np.nonzero(b[q])[0]
            obs=np.random.choice(bq)
            sequences.append(reverse_sequences_map[obs])
            if obs == 0:
```

```
                break
            aq = np.nonzero(a[q])[0]
            q = np.random.choice(aq)

        gen_sequences.append(sequences)
    return gen_sequences

generating_sequences = gen_sequences(b_1,a_1)
print("Generated sequences are”)
for i in generating_sequences:
    print(i,end="\n")
```

```
Generated sequences are
['A', 'C', 'D', 'C', 'C', 'D', 'S']
['B', 'C', 'S']
['B', 'C', 'A', 'D', 'B', 'B', 'B', 'C', 'C', 'S']
['B', 'C', 'S']
['B', 'D', 'B', 'D', 'B', 'D', 'S']
['A', 'C', 'C', 'B', 'C', 'B', 'A', 'D', 'S']
['B', 'C', 'S']
['B', 'C', 'C', 'C', 'B', 'D', 'C', 'S']
['A', 'D', 'A', 'A', 'D', 'C', 'A', 'B', 'C', 'S']
['B', 'D', 'B', 'D', 'D', 'C', 'C', 'A', 'D', 'B', 'D', 'A', 'D',
'C', 'B', 'D', 'D', 'D', 'S']
```

Part b - Implementing the forward algorithm to classify the given sequences

Code:

```
def forward_hmm_prob(pi,a,b,N,obs):
    T = len(obs)
    forward = np.zeros((N,T))
    for s in states:
        forward[s,0] = pi[s] * b[s][obs[0]]

    for t in range(1,T):
        for s in range(0,N):
            sm = 0
            for k in range(0,N):
                sm += forward[k,t-1] * a[k][s] * b[s][obs[t]]
            forward[s,t] = sm

    forward_prob = np.sum(forward[:,T-1])
    return forward_prob


sequences = [['A','D','C','B','D','C','C','S'],
             ['B','D','S'],
```

```
             ['B','C','C','B','D','D','C','A','C','S'],
             ['A','C','D','S'],
             ['A','D','A','C','S'],
             ['D','B','B','S'],
             ['A','B','S'],

['D','D','B','D','D','B','A','C','C','D','A','B','B','C','D','B','
B','B','S'],
             ['D','B','D','S'],
             ['A','A','A','A','D','C','B','S']
            ]

N = 4

for sequence in sequences:
    new_seq = [sequences_map[i] for i in sequence]
    forward_hmm_1_prob = forward_hmm_prob(pi_1,a_1,b_1,N,new_seq)
    forward_hmm_2_prob = forward_hmm_prob(pi_2,a_2,b_2,N,new_seq)
    if forward_hmm_1_prob > forward_hmm_2_prob:
        print("HMM 1")
    else:
        print("HMM 2")
```

Output

```
HMM 1
HMM 1
HMM 1
HMM 1
HMM 1
HMM 2
HMM 2
HMM 2
HMM 2
HMM 2
```

Part 3 - Implementing the viterbi algorithm to decode the hidden states for the given observations

Code:

```
def viterbi_hmm(pi,a,b,N,obs):
    T = len(obs)
    viterbi = np.zeros((N,T))
    backpointer = np.zeros((N,T),dtype=int)
    for s in states:
        viterbi[s,0] = pi[s] * b[s][obs[0]]
        backpointer[s,0] = 0
```

```
    for t in range(1,T):
        for s in range(N):
            max_arr = np.array([])
            backpointer_max = np.array([])
            for k in range(0,N):
                max_arr = np.append(max_arr,viterbi[k,t-1] * a[k]
[s] * b[s][obs[t]])
                backpointer_max = 
np.append(backpointer_max,viterbi[k,t-1] * a[k][s] * b[s][obs[t]])
            viterbi[s,t] = max(max_arr)
            backpointer[s,t] = np.argmax(backpointer_max)

    bestpathprob = np.max(viterbi[:,T-1])
    bestpathpointer = np.argmax(viterbi[:,T-1])

    qt_s = int(bestpathpointer)

    j = T-1
    best_path = []
    while j >= 0:
        best_path.append(qt_s+1)
        qt_s = int(backpointer[qt_s][j])
        j -=1
    best_path.reverse()
    return best_path,bestpathprob

for sequence in sequences:
    print("Sequence",sequence)
    new_seq = [sequences_map[i] for i in sequence]
    print(viterbi_hmm(pi_2,a_2,b_2,N,new_seq)[0])
```

Output

```
Sequence ['A', 'D', 'C', 'B', 'D', 'C', 'C', 'S']
[4, 2, 3, 2, 2, 3, 3, 1]
Sequence ['B', 'D', 'S']
[1, 1, 1]
Sequence ['B', 'C', 'C', 'B', 'D', 'D', 'C', 'A', 'C', 'S']
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Sequence ['A', 'C', 'D', 'S']
[4, 3, 2, 1]
Sequence ['A', 'D', 'A', 'C', 'S']
[4, 4, 4, 3, 1]
Sequence ['D', 'B', 'B', 'S']
[4, 2, 3, 1]
Sequence ['A', 'B', 'S']
[4, 2, 1]
```

```
Sequence ['D', 'D', 'B', 'D', 'D', 'B', 'A', 'C', 'C', 'D', 'A',
'B', 'B', 'C', 'D', 'B', 'B', 'B', 'S']
[4, 2, 3, 2, 2, 3, 4, 3, 3, 4, 4, 2, 2, 3, 2, 3, 2, 3, 1]
Sequence ['D', 'B', 'D', 'S']
[4, 2, 2, 1]
Sequence ['A', 'A', 'A', 'A', 'D', 'C', 'B', 'S']
[4, 4, 4, 4, 2, 3, 2, 1]
```

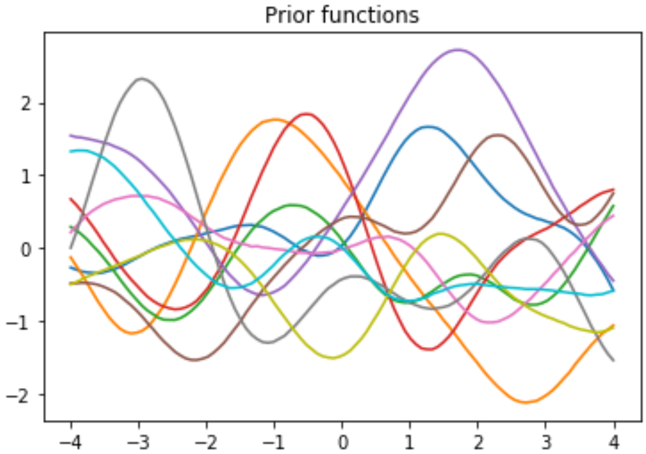
Solution 4

Code

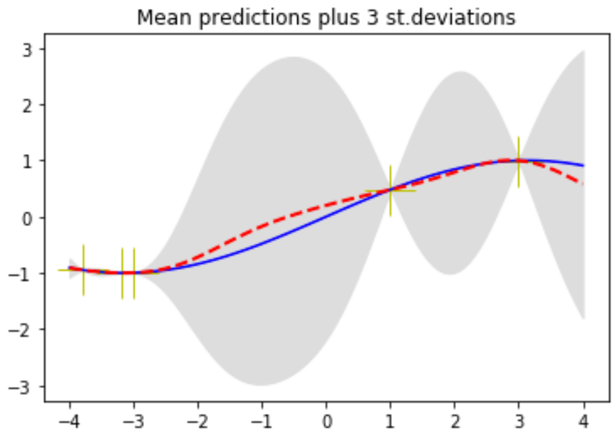
```
import numpy as np
import matplotlib.pyplot as plt

f = lambda x: np.sin(0.5*x).flatten()

def kernel(x,y):
    #Kernel function (RBF)
    sq_dist = np.sum(x**2,1).reshape(-1,1) + np.sum(y**2,1) -
2*np.dot(x,y.T)
    return np.exp(-.5*sq_dist)

n = 100 # number of test points
N = 5 # number of training points

s = 0.0001 #noise

#given training points
D = [(-3.8,-0.9463),(-3.2,-0.9996),(-3,-0.9975),(1,0.4794),
(3,0.9975)]

X = [d[0] for d in D]
y = [d[1] for d in D]

Xtrain = np.array(X).reshape(5,1)
ytrain = f(Xtrain) + s*np.random.randn(N)

Xtest =  np.linspace(-4,4,n).reshape(-1,1)
ytest = f(Xtest)


#drawing samples from prior at the test points - Check this
expression why is that what it is
Kss = kernel(Xtest,Xtest)

L = np.linalg.cholesky(Kss + s * np.eye(n))
f_prior = np.dot(L,np.random.normal(size=(n,10)))

plt.title('Prior functions')
plt.plot(Xtest,f_prior)


#NonLinear Regression

#covariance between training points
K = kernel(Xtrain,Xtrain)
L = np.linalg.cholesky(K + s * np.eye(N))
```

```
#Compute the mean and variance of the test points

# Using linalg.solve to solve the system of linear equations for
K_star
Lk = np.linalg.solve(L, kernel(Xtrain,Xtest))
mu = np.dot(Lk.T, np.linalg.solve(L, ytrain))

#taking only the digonal values from covariance matrix for getting
the standard deviation
s2 = np.diag(Kss) - np.sum(Lk**2,axis = 0)
s = np.sqrt(s2)


#Plots for regression

#Plot for mean points on the top of test distribution

#Plotting the training distribution
plt.plot(Xtrain,ytrain,'y+',ms=20)

# #Plotting the test distribution (Xtest, ytest)
print(Xtest.shape,ytest.shape)
plt.plot(Xtest,ytest,'b-')

#Plotting the confidence interval of the distribution
plt.gca().fill_between(Xtest.flat, mu-3*s, mu+3*s,
color="#dddddd")

#Plotting the mean points on the test distribution
plt.plot(Xtest,mu,'r--',lw=2)
plt.title('Mean predictions plus 2 st.deviations’)



# plotting the posterior distribution functions

#Using the GP regression algorithm that uses variance to plot the
posterior functions derived using gaussian conditional approach
L = np.linalg.cholesky(Kss + s*np.eye(n) - np.dot(Lk.T,Lk))
f_post = mu.reshape(-1,1) + np.dot(L,np.random.normal(size = (n,
10)))
plt.title('Ten samples from GP posterior')
plt.plot(Xtest,f_post)
```

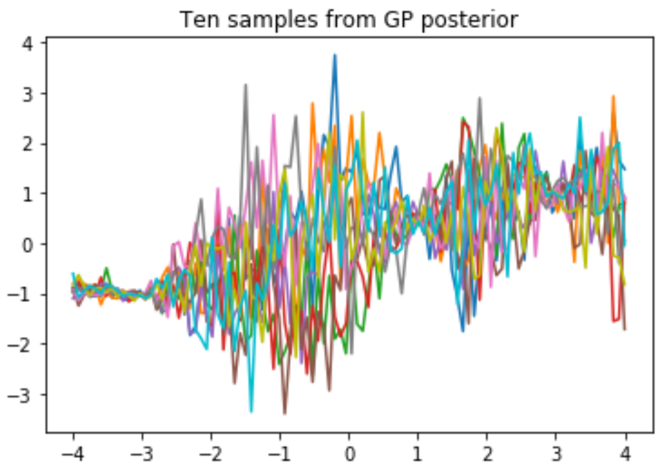
Plots



Prior functions



Mean predictions plus 3 st.deviations



Ten samples from GP posterior