Worked with: Steve Novakov

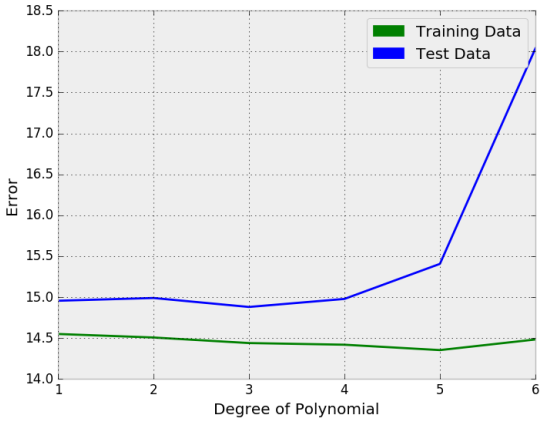
1 Problem 1

1.1 Part a

For both of these sections, I used the same code, which would output 2 plots, shown below. Once the plots are shown, I will show my code which generated both plots. The reason we need to disregard the id number is that it has no correlation to the controller input. It is simply a naming convention which allows us to keep track of the samples, not something that contributes to any outcome of the data. The id could just have easily been letters, roman numerals, or any other naming convention.

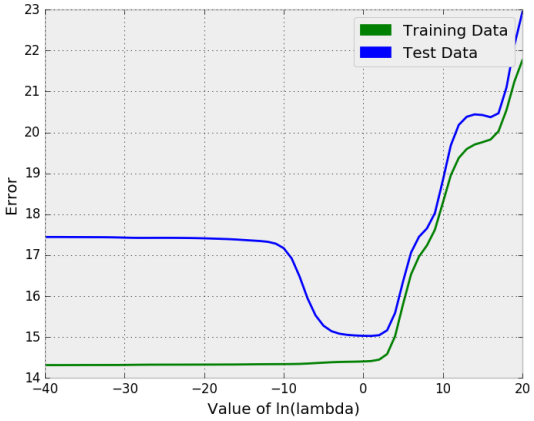
1.1.1 Part i

The plot of RMSE against M is shown below



1.1.2 Part ii

The plot of RMSE against λ is shown below The code for both of these



1.1.3 Code

The code, with comments, is shown below

```
import numpy as np
import pandas
import math
import matplotlib.patches as mpatches
from matplotlib import pyplot as plt;
if "bmh" in plt.style.available: plt.style.use("bmh");

#Import the data
Data = pandas.read_csv("train_graphs_f16_autopilot_cruise.csv", sep=",",
                           names= ["id","rolling_speed","elevation_speed","elevation_jerk",
"elevation,roll","elevation_acceleration","controller_input"])

Test = pandas.read_csv("test_graphs_f16_autopilot_cruise.csv", sep=",",
                           names= ["id","rolling_speed","elevation_speed","elevation_jerk",
"elevation,roll","elevation_acceleration","controller_input"])


names= ["rolling_speed","elevation_speed","elevation_jerk",
"elevation,roll","elevation_acceleration"]


#Shave off the id section
Test = Test[1:]

Error_P=[]
Error_R=[]
Error_Reg_Test=[]
Error_Test=[]
ErrorR_test = []

#Max degree of polynomial
j=6
d = range(j+1)
#Lambda values
lamb = range(-40,21,1)


t2=[]
print d[1:]

for n in d[1:]:
    Theta=[]
    ThetaT=[]
    m=0
    t=[]
```

```
#Generate the Theta Matrix for the Training data
for x in range(1,len(Data)+1):
    t.append(float(Data["controller_input"][x]))
    row = [1]
    for y in range(len(names)):
        for z in range(1,n+1):
            Value = float(Data[names[y]][x])
            row.append(Value**(z))
    Theta.append(row)
#Generate the Theta Matrix for the Test data
for x in range(len(Test)):
    t2.append(float(Test["controller_input"][x]))
    row = [1]
    for y in range(len(names)):
        for z in range(1,n+1):
            Value = float(Test[names[y]][x])
            row.append(Value**(z))
    ThetaT.append(row)
#Compute w without regularization
m = len(np.dot(np.transpose(Theta),Theta))
w = np.dot(np.dot(np.linalg.inv(np.dot(np.transpose(Theta),Theta)),np.transpose(Theta)),t)

#Compute w with regularization for n=6
if n==6:
    for lam in lamb:
        ErrorR=0
        ErrorR_test=0
        lam = np.exp(lam)
        wR = np.dot(np.dot(np.linalg.inv(np.dot(np.transpose(Theta),Theta)
        +lam*np.identity(m)),np.transpose(Theta)),t)
        for x in range(len(Data)):
            Error2 = (np.dot(Theta[x],wR)-t[x])**2#+lam/2*np.dot(np.transpose(wR),wR)
            ErrorR = ErrorR+Error2
        for x in range(len(Test)):
            Error2 = (np.dot(ThetaT[x],wR)-t2[x])**2#+lam/2*np.dot(np.transpose(wR),wR)
            ErrorR_test = ErrorR_test+Error2
        Error_R.append(np.sqrt(ErrorR/len(Data)))
        Error_Reg_Test.append(np.sqrt(ErrorR_test/len(Test)))

#U,S,V= np.linalg.svd(np.dot(np.transpose(Theta),Theta))

#Find the test and training error for the two examples
ErrorT=0
Error_T=0
for x in range(len(Data)):
    Error = (np.dot(Theta[x],w)-t[x])**2
    ErrorT = ErrorT+Error
for x in range(len(Test)):
```

```
        Error_T2 = (np.dot(ThetaT[x],w)-t2[x])**2
        Error_T= Error_T+Error_T2
    Error_P.append(np.sqrt(ErrorT/len(Data)))
    Error_Test.append(np.sqrt(Error_T/len(Test)))

#plot the results
print Error_P
print Error_Test
print Error_Reg_Test
print d[1:]
plt.plot(d[1:],Error_P,"g",d[1:],Error_Test,"b")
plt.ylabel('Error')
plt.xlabel('Degree of Polynomial')
green_patch = mpatches.Patch(color='green', label='Training Data')
blue_patch = mpatches.Patch(color='blue', label='Test Data')
plt.legend(handles=[green_patch, blue_patch])
plt.show()

plt.plot(lamb,Error_R,"g",lamb,Error_Reg_Test,"b")
plt.ylabel('Error')
plt.xlabel('Value of ln(lambda)')
green_patch = mpatches.Patch(color='green', label='Training Data')
blue_patch = mpatches.Patch(color='blue', label='Test Data')
plt.legend(handles=[green_patch, blue_patch])
plt.show()
```

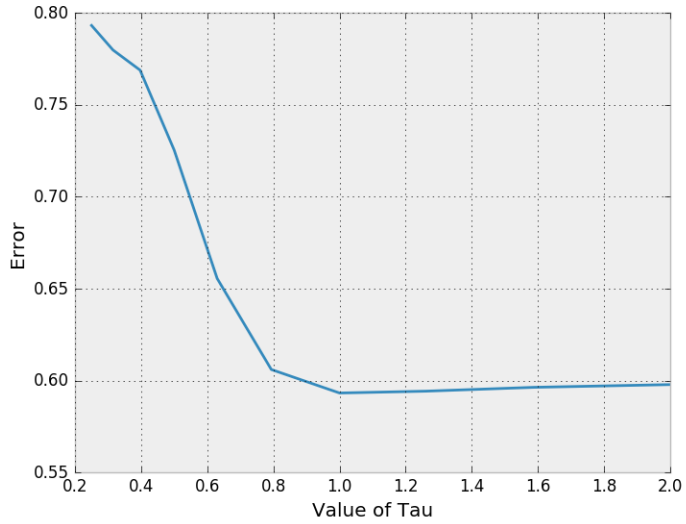
1.2 Part 1b

In this section, we were asked to consider locally weighted linear regression. For this example, I normalized all of the data. This means, for a particular data column x_j, each term in the column was normalized via

$$x_{ij} = \frac{(x_{ij} - \mu_{x_j})}{\sigma_{x_j}} \tag{1}$$

This was done for each training and test point. As a note, the test points were normalized by their respective training data columns. This means the mean of training column 1 was subtracted from the test column 1 value, then divided by the standard deviation of training data column 1 to get the normalized test data. With this being said, the final plot is shown below: As a note, I had to delete off the names row in the training data set in order for the code to run. Otherwise, the mean function tried to take the mean of string vectors which would cause problems. Once this id vector was deleted, the code ran fine. The code, with comments, is shown below:

```
import numpy as np
import pandas
import math
import scipy.sparse as sp
import scipy.sparse.linalg as spl
from matplotlib import pyplot as plt;
if "bmh" in plt.style.available: plt.style.use("bmh");
```




```
#Get data
train = pandas.read_csv("train_graphs_f16_autopilot_cruise.csv", sep=",",
                            names= ["id","rolling_speed","elevation_speed","elevation_jerk",
"elevation","roll","elevation_acceleration","controller_input"])

test = pandas.read_csv("test_locreg_f16_autopilot_cruise.csv", sep=",",
                            names= ["id","rolling_speed","elevation_speed","elevation_jerk",
"elevation","roll","elevation_acceleration","controller_input"])

names= ["rolling_speed","elevation_speed","elevation_jerk",
"elevation","roll","elevation_acceleration"]


#Slice off the id from the test data
Train = train
Test  = test[1:]

Error= 0
ErrorT =[]
Theta = []
t = []

#Find the mean and standard deviation of training data
mT= Train.mean()
SDT= Train.std()

#Generate theta and the parameter value vector for the training data
for z in range(len(Train)):
    row =[1]
    for name in names:
```

```
        mN = mT[name]
        SD = SDT[name]
        Value = (float(Train[name][z])-mN)/SD
        row.append(Value)
    Theta.append(row)
    c = "controller_input"
    ti = (Train[c][z]-mT[c])/SDT[c]
    t.append(float(ti))


Tau = np.logspace(-2, 1, num=10, base=2)


for tau in Tau:
    Error=0
    #Cycle through each test point
    for y in range(1,len(Test)+1):
        r=[]
        R=[]
        #Cycle through the training data
        for z in range(len(Train)):
            x  = []
            xi = []
            dif = []
            X   = [1]
            #Normalize each value and compute local weight
            for name in names:
                mN = mT[name]
                SD = SDT[name]
                x1 = (Train[name][z]-mN)/SD
                x2 = (float(Test[name][y])-mN)/SD
                x.append(float(x2))
                xi.append(float(x1))
                X.append(float(x2))
                dif.append(float(x1)-float(x2))
            nm = (np.dot(np.transpose(dif),dif))
            ri = np.exp(-nm/(2*(float(tau)**2)))
            r.append(float(ri))

        #Generate R
        R = np.diag(r)

        #Find the w vector and find the error
        TT = np.transpose(Theta).dot(R)
        TT = np.dot(TT,Theta)
        t  = np.asarray(t)
        w1  = np.dot(np.dot(np.linalg.inv(TT),np.transpose(Theta)),R)
        w   = np.dot(w1,t)
```

```
        tt  = (float(Test["controller_input"][y])-mT[c])/SDT[c]
        Er  =(np.dot(np.transpose(w),X)-float(tt))**2/len(Test)
        Error = Error+Er
    #Error for each value of Tau
    ErrorT.append(np.sqrt(Error))


#Plot the results
plt.plot(Tau,ErrorT)
plt.ylabel('Error')
plt.xlabel('Value of Tau')
plt.show()
```

2 Problem 2

The code I used is shown below. As a note, the tuning variables I used was the optimal polynomial degree (n_{opt}) and λ for the regularization.

```
import numpy as np
import pandas
import math

#Import Data
messages = pandas.read_csv("steel_composition_train.csv", sep=",",
                           names=["id", "Carbon","Nickel","Manganese","Sulfer",
                           "Chromium","Iron","Phosphorus","Silicon","Strength"])

names=["Carbon","Nickel","Manganese","Sulfer","Chromium","Iron","Phosphorus","Silicon"]
L    = len(messages)/2
Data = messages[1:L]
Test = messages[L:]

#print L



Error_P=[]
Error_R=[]
Error_Reg_Test=[]
Error_Test=[]
ErrorR_Test=[]

#Maximum polynomial degree
j=6
d = range(j+1)
#Lambda range
lamb = range(-100,10,1)
```

```
t2=[]
nopt=5


for n in d[1:]:
    Theta=[]
    ThetaT=[]
    m=0
    t=[]
    #Cut the training data in half. First half is for training, second for test
    for x in range(1,len(Data)+1):
        t.append(float(Data["Strength"][x]))
        row = [1]
        #Generate Theta for training data
        for y in range(len(names)):
            for z in range(1,n+1):
                Value = float(Data[names[y]][x])
                row.append(Value**(z))
        Theta.append(row)
    for x in range(len(Test)):
        t2.append(float(Test["Strength"][L+x]))
        row = [1]
        #Generate Theta for test data
        for y in range(len(names)):
            for z in range(1,n+1):
                Value = float(Test[names[y]][L+x])
                row.append(Value**(z))
        ThetaT.append(row)

    #Length of identity for regularization
    m = len(np.dot(np.transpose(Theta),Theta))
    #Compute w vector
    w = np.dot(np.dot(np.linalg.inv(np.dot(np.transpose(Theta),Theta)),np.transpose(Theta)),t)

    #Pick the degree, then cycle through lambda to find best lambda
    if n==nopt:
        for lam in lamb:
            ErrorR=0
            ErrorR_test=0
            lam = np.exp(lam)
            wR = np.dot(np.dot(np.linalg.inv(np.dot(np.transpose(Theta),Theta)
            +lam*np.identity(m)),np.transpose(Theta)),t)
            #Compute the regularized error
            for x in range(len(Data)):
                Error2 = (np.dot(Theta[x],wR)-t[x])**2#+lam/2*np.dot(np.transpose(wR),wR)
                ErrorR = ErrorR+Error2
            for x in range(len(Test)):
```

```
                Error2 = (np.dot(ThetaT[x],wR)-t2[x])**2#+lam/2*np.dot(np.transpose(wR),wR)
                ErrorR_test = ErrorR_test+Error2
            Error_R.append(np.sqrt(ErrorR/len(Data)))
            Error_Reg_Test.append(np.sqrt(ErrorR_test/len(Test)))
            Val = np.sqrt(ErrorR_test/len(Test))
            #Find the minimum
            if (np.amin(Error_Reg_Test)==Val):
                lam_min = lam

    U,S,V= np.linalg.svd(np.dot(np.transpose(Theta),Theta))

    #Find the test and training error for the non-regularized Least Squares
    ErrorT=0
    Error_T=0
    for x in range(len(Data)-1):
        Error = (np.dot(Theta[x],w)-t[x])**2
        ErrorT = ErrorT+Error
    for x in range(len(Test)):
        Error_T2 = (np.dot(ThetaT[x],w)-t2[x])**2
        Error_T= Error_T+Error_T2
    Error_P.append(np.sqrt(ErrorT/len(Data)))
    Error_Test.append(np.sqrt(Error_T/len(Test)))

#Output the results
print "The train least squares Errors are ",Error_P
print "..."
print "..."
print "..."
print "The train Regularized Errors are ",Error_R
print "..."
print "..."
print "..."
print "The test least squares Errors are ",Error_Test
print "..."
print "..."
print "..."
print "The test Regularized Errors are ",Error_Reg_Test
print "..."
print "..."
print "..."
print "The minimum train error is ",np.amin(Error_P)
print "..."
print "..."
print "..."
print "The minimum train Regularized Errors is ",np.amin(Error_R)
print "..."
print "..."
print "..."
```

```
print "The minimum test least squares Errors is ",np.amin(Error_Test)
print "..."
print "..."
print "..."
print "The minimum test Regularized Errors is ",np.amin(Error_Reg_Test)

print lam_min

#set lambda to the minimum value above
lam=lam_min

#Now load all of the training data
messages = pandas.read_csv("steel_composition_train.csv", sep=",",
                           names=["id", "Carbon","Nickel","Manganese","Sulfer",
                           "Chromium","Iron","Phosphorus","Silicon","Strength"])

names=["Carbon","Nickel","Manganese","Sulfer","Chromium","Iron","Phosphorus","Silicon"]
Data = messages[1:]

#Compute the Theta for all of the training data, not half

Error_P=[]

for n in range(j):
    Theta=[]
    m=0
    t=[]

    for x in range(len(Data)-1):
        t.append(float(Data["Strength"][x+1]))
        row = [1]
        for y in range(len(names)):
            for z in range(nopt):
                Value = float(Data[names[y]][x+1])
                row.append(Value**(z+1))

        Theta.append(row)
lam=1
    m = len(np.dot(np.transpose(Theta),Theta))
    w = np.dot(np.dot(np.linalg.inv(np.dot(np.transpose(Theta),Theta)
            +lam*np.identity(m)),np.transpose(Theta)),t)

    ErrorT=0

    for x in range(len(Data)-1):
        Error = (np.dot(Theta[x],w)-t[x])**2
        ErrorT = ErrorT+Error
    Error_P.append(np.sqrt(ErrorT/len(Data)))
```

```
    if n==nopt:
        W=w
print Error_P
print W
print nopt

#Import the test data
messages = pandas.read_csv("steel_composition_test.csv", sep=",",
                           names=["id", "Carbon","Nickel","Manganese","Sulfer",
                           "Chromium","Iron","Phosphorus","Silicon","Strength"])


TEST = messages[1:]



Theta=[]
Strength=[]
N=nopt

#Solve for strength with your optimal polynomial and lambda
for x in range(len(TEST)):
    row = [1]
    for y in range(len(names)):
        for z in range(N):
            Value = float(TEST[names[y]][x+1])
            row.append(Value**(z+1))
    Theta.append(row)
Strength=(np.dot(Theta,W))


#Write to a Dataframe
df = pandas.DataFrame(Strength, range(1,413),columns=['Strength'])

df.to_csv('erwaner_test_data_n=5_lam=3000.csv')
```

3 Problem 3

For this problem, we want to minimize

$$E_D(w) = \frac{1}{2}\sum_{i=1}^{N} r_i(w^T x_i - t_i)^2 \tag{2}$$

3.1 Part a

We are asked to show that $E_D(w)$ can be written as

$$E_D(w) = (Xw - t)^T R(Xw - t) \tag{3}$$

To do so, we start at the original definition, i.e.,

$$E_D(w) = \frac{1}{2}\sum_{i=1}^{N} r_i(w^T x_i - t_i)^2 \tag{4}$$

which can be decomposed as:

$$E_D(w) = (w^T x_1 - t_1)r_1(w^T x_1 - t_1) + (w^T x_2 - t_2)r_2(w^T x_2 - t_2) + \cdots + (w^T x_N - t_N)r_N(w^T x_N - t_N) \tag{5}$$

As a note, the term r_i can be moved between the two $(w^T x_1 - t_1)$ terms because all three terms are scalar, so they can be moved around as desired. With this said, this can be further equated to

$$\begin{aligned} E_D(w) = \frac{1}{2}\Bigg(\begin{bmatrix} x_{11} & \dots & x_{1n} \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} - t_1 \Bigg) r_1 \Bigg(\begin{bmatrix} x_{11} & \dots & x_{1n} \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} - t_1 \Bigg) + \dots \\ + \Bigg(\begin{bmatrix} x_{N1} & \dots & x_{Nn} \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} - t_N \Bigg) r_N \Bigg(\begin{bmatrix} x_{N1} & \dots & x_{Nn} \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} - t_N \Bigg) \end{aligned}$$

defining $e_i = (w^T x_i - t_i)$, the above becomes

$$e_1 r_1 e_1 + \cdots + e_N r_N e_N \tag{6}$$

From class, we know

$$\Phi = \begin{bmatrix} x_{11} & \dots & x_{1n} \\ \vdots & & \vdots \\ x_{N1} & \dots & x_{Nn} \end{bmatrix} \tag{7}$$

Due to the like terms w, the above can be reformulated as

$$E_D(w) = \left(\begin{bmatrix} x_{11} & \dots & x_{1n} \\ \vdots & & \vdots \\ x_{N1} & \dots & x_{Nn} \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} - \begin{bmatrix} t_1 \\ \vdots \\ t_N \end{bmatrix} \right)^T \begin{bmatrix} \frac{r_1}{2} & 0 & \dots & & 0 \\ 0 & \frac{r_2}{2} & 0 & \dots & 0 \\ \vdots & & & & \vdots \\ 0 & \dots & & & \frac{r_N}{2} \end{bmatrix} \left(\begin{bmatrix} x_{11} & \dots & x_{1n} \\ \vdots & & \vdots \\ x_{N1} & \dots & x_{Nn} \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} - \begin{bmatrix} t_1 \\ \vdots \\ t_N \end{bmatrix} \right) \tag{8}$$

Decomposing the above, we get

$$\left(\begin{bmatrix} x_{11}w_1 + \cdots + x_{1n} \\ \vdots \\ x_{N1}w_1 + \cdots + x_{Nn}w_n \end{bmatrix} - \begin{bmatrix} t_1 \\ \vdots \\ t_N \end{bmatrix}\right)^T \begin{bmatrix} \frac{r_1}{2} & 0 & \cdots & & 0 \\ 0 & \frac{r_2}{2} & 0 & \cdots & 0 \\ \vdots & & & & \vdots \\ 0 & \cdots & & & \frac{r_N}{2} \end{bmatrix} \left(\begin{bmatrix} x_{11}w_1 + \cdots + x_{1n} \\ \vdots \\ x_{N1}w_1 + \cdots + x_{Nn}w_n \end{bmatrix} - \begin{bmatrix} t_1 \\ \vdots \\ t_N \end{bmatrix}\right) \tag{9}$$

$$= \begin{bmatrix} e_1 & \cdots & e_N \end{bmatrix} \begin{bmatrix} \frac{r_1}{2} & 0 & \cdots & & 0 \\ 0 & \frac{r_2}{2} & 0 & \cdots & 0 \\ \vdots & & & & \vdots \\ 0 & \cdots & & & \frac{r_N}{2} \end{bmatrix} \begin{bmatrix} e_1 \\ \vdots \\ e_N \end{bmatrix} \tag{10}$$

$$= \frac{1}{2}\left(e_1 r_1 e_1 + e_2 r_2 e_2 + \cdots + e_N r_N e_N\right) \tag{11}$$

Which is exactly what we started with. Therefore, I can conclude

$$X = \begin{bmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & & \vdots \\ x_{N1} & \cdots & x_{Nn} \end{bmatrix}, \quad t = \begin{bmatrix} t_1 \\ \vdots \\ t_N \end{bmatrix}, \quad R = \begin{bmatrix} \frac{r_1}{2} & 0 & \cdots & & 0 \\ 0 & \frac{r_2}{2} & 0 & \cdots & 0 \\ \vdots & & & & \vdots \\ 0 & \cdots & & & \frac{r_N}{2} \end{bmatrix} \tag{12}$$

As a note, n is the length of x_i (and consequently w). I.e., this is the length of the specified least squares vector. N is the amount of samples we are considering. Therefore, in this example $X \in \mathbb{R}^{N\times n}$, $t \in \mathbb{R}^{N\times 1}$, and $R \in \mathbb{R}^{N\times N}$. In this definition X, R are matrices and t is a vector.

3.2 Part b

So we know

$$E_D(w) = (Xw - t)^T R(Xw - t) \tag{13}$$

Which, expanding out, is equivalent to:

$$E_D(w) = w^T X^T RXw - w^T X^T Rt - t^T RXw + t^T Rt \tag{14}$$

Taking $\bigtriangledown_w E_D(w)$,we get

$$\bigtriangledown_w E_D(w) = 2X^T RXw - 2X^T Rt = 0 \tag{15}$$

And we get the equation

$$X^T RXw = X^T Rt \tag{16}$$

$$w = (X^T RX)^{-1} X^T Rt \tag{17}$$

Also, to compute this w, we must ensure that $X^T RX$ is invertible. If the objective function is convex, we can say $X^T RX$ is invertible because finding where the gradient is zero finds the global minimizer. Additionally, we know that the sum of convex function is convex, so we must simply prove $r_i(w^T x_i - t_i)^2$ is convex for all x. Since this is a well known convex function (i.e., y^2 when $y = w^T x - t$), we can say the sum of these convex functions is convex and there is a global minimizer. Therefore, $w = (X^T RX)^{-1} X^T Rt$ finds this global minimizer.

3.3 Part c

We are given

$$p(t_i|x_i;w) = \frac{1}{\sqrt{2\pi}\sigma_i} exp\left(-\frac{(t_i - w^T x_i)^2}{2(\sigma_i)^2}\right) \tag{18}$$

To find the maximum likelihood estimator, we take $E(w) = -log(p(t|x;w))$, which amounts to

$$E(w) = -log\left(\sum_{i=1}^{N} \frac{1}{\sqrt{2\pi}\sigma_i} exp\left(-\frac{(t_i - w^T x_i)^2}{2(\sigma_i)^2}\right)\right) \tag{19}$$

Pulling the log and negative sign in, and decomposing due to log rules, we get

$$E(w) = \sum_{i=1}^{N} \left(log(2\pi) + log(\sigma_i) + \frac{(t_i - w^T x_i)^2}{2(\sigma_i)^2}\right) \tag{20}$$

Taking $\bigtriangledown_w E(w)$,we get

$$\bigtriangledown_w E(w) = \frac{\delta}{\delta w}\left(\sum_{i=1}^{N} \frac{(t_i - w^T x_i)^2}{2(\sigma_i)^2}\right) = 0 \tag{21}$$

Which is the same equation as we just solved in part a and b, with $r_i = 1/\sigma_i^2$. Additionally, the numerator is $t_i - w^T x_i$ instead of $w^T x_i - t_i$. However, this difference will work itself out, since the value is being squared, so this difference will not ultimately matter.

4 Problem 4

4.1 Part a

4.1.1 i

In the pre-processing step, we take the median of each of the 57 input features, then either map them to 1 or 2. If the features are greater than the median of their column, they are mapped to 2. Otherwise, the data point is mapped to 1. This was done for both the training and test data. I do believe our data is a ratio variable, although it does not have a clear definition of zero the value of 1 can be thought of as our "zero". Ratio variables can also be added, subtracted, multiplied, and divided, which our variables clearly can do. An ordinal variable is a variable where order matters but not the difference between values. In our case, we only have 1 difference (between 2 and 1) and therefore the difference does matter, as it corresponds to being above or below the mean. Additionally, ordinal variables are typcially measures of non-numeric concepts, which cleary is not the case for our data. I do not believe our data is additionally not a nominal variable. Nominal variables are used for labeling variables without any quantitative value. It is pretty clear that we are using our variables for quantitative values and probabilistic characteristics. I also believe that our data is an interval variable, since the difference between two values is meaningful. A value of 2 when compared to 1 is meaningful for our calculations.

Our data set has measures of: percentage or words, percentage of characters that match CHAR, average length of uninterrupted sequences of capital letters, longest uninterrupted sequence of capital letters, sum of length of uninterrupted sequences of capital letters, and total number of capital letters in the email. I believe that the first category, the percentage of words that match WORD, is suitable for pre-processing. There should not be many outliers, because there are a large number of words, so things will start to even out. Outliers will have less of an effect and the median will overall be a more accurate way to divide the data. The same logic will hold for the percentage of characters in the email. There will be so many letters that outliers will have a small effect. However, for the last three categories I think there could possibly be some error in the pre-processing step. There, theoretically, won't be that many capital letters, most normal emails really only have at most 1 capital letter per word, if that. Therefore, a large number of outliers with a lot of capital letters can skew the median to something much larger than an accurate divide would be. The will be the case for all three of the last categories, which can all be skewed by a few emails entirely in Caps or a very long word in all Caps. The one that would be the least skewed is the first category, which is the average length of uninterrupted sequences of capital letters. Although I would still think this could be skewed by outliers.

4.1.2 ii

For this section, I used the Naive Bayes algorithm in its original form, which assumes independence. This means my algorithm is determined by:

$$P(x_i, y_i = c) = P(y_i = c) \prod_{j=1}^{D} P\left(x_{ij} \middle| (y_i = c)\right) \tag{22}$$

and the category (normal, spam) is determined by the larger of $P(x_i, y_i = normal)$ and $P(x_i, y_i = spam)$. This is shown in the code attached, with the results shown below As you can see, the Naive Bayes algorithm correctly predicts the type of email in the test data around 89% of the time. By comparison, the test data is around 61% normal emails. Therefore, had I just chosen normal emails

```
The misclassification percentage is  0.111111111111
The percentage of spam is  0.385620915033
The percentage of normal email is  0.614379084967
If I had guessed normal email every time, I would be correct  0.614379084967  of
 the time
Therefore the algorithm improves the results  0.274509803922  percent
```

I would have been correct 61% of the time. Therefore, my Naive Bayes algorithm is around 27.5% better than just guessing the majority category (normal).

The code for the above problem is shown below, with attached comments:

```
import numpy as np
import pandas
from collections import Counter
import math
import re

#import the data
train = pandas.read_csv("spambase.train", sep=',',
                     names=np.array(map(str,range(58))))

test = pandas.read_csv("spambase.test", sep=',',
                    names=np.array(map(str,range(58))))

nums = range(57)
names = np.array(map(str,nums))

#Set values above the median to 2, and below to 1
for col in names:

            med = float(np.median(train[col]))

            column = train[col]
            col2 = column-med
            column.loc[col2>0]=2
            column.loc[col2<=0]=1
            train.loc[:,col]=column

            column2 = test[col]
            col3 = column2-med
            column2.loc[col3>0]=2
            column2.loc[col3<=0]=1
            test.loc[:,col]=column2

#Separate into Normal and Spam
Normal = train[train["57"]==0]
Spam   = train[train["57"]==1]

#Find Probabilities of Normal and Spam
PN = float(len(Normal))/float(len(train))
```

```
PS = float(len(Spam))/float(len(train))

print PN
print PS
P1GN=dict()
P2GN=dict()
P1GS=dict()
P2GS=dict()
P1 = dict()
P2 = dict()

#Compute the conditional probabilities
for col in names:
           P1GN[col] = float(len(Normal[Normal[col]==1]))/float(len(Normal))
           P2GN[col] = float(len(Normal[Normal[col]==2]))/float(len(Normal))
           P1GS[col] = float(len(Spam[Spam[col]==1]))/float(len(Spam))
           P2GS[col] = float(len(Spam[Spam[col]==2]))/float(len(Spam))
P1   = float(len(train[train[col]==1]))/float(len(train))
P2   = float(len(train[train[col]==2]))/float(len(train))

Decision = []
Error    = []

#Compute the Naive Bayes
for line in range(len(test)):
           PN1 = P1
           PS1 = P2
           for col in names:
                     C= test[col]
                     if test[col][line]==1:
                                PN1 = PN1*P1GN[col]
                                PS1 = PS1*P1GS[col]
                     else:
                                PN1 = PN1*P2GN[col]
                                PS1 = PS1*P2GS[col]
           EV = PN*PN1+PS*PS1
           PNormal = PN1/EV
           PSpam   = PS1/EV
           if PNormal>PSpam:
                     d=0
                     Decision.append(d)
           else:
                     d=1
                     Decision.append(d)
           Error.append(abs(d-test["57"][line]))

print "The misclassification percentage is ", float(sum(Error))/float(len(test))
print "The percentage of spam is ",float(sum(test["57"]))/float(len(test))
```

```
print "The percentage of normal email is ",1.0-float(sum(test["57"]))/float(len(test))
print "If I had guessed normal email every time, I would be correct ", 1.0-float(sum(test["57"]
print "Therefore the algorithm improves the results ",(1.0-float(sum(Error))/float(len(test)))
```

4.2 Part b

For this problem, we are asked to briefly describe how to extract features from text. To extract features, I first separated the data into the two categories, ham and spam. With each email siphoned into its appropriate category, I could easily compute the probability of each of the classes, i.e. P(spam) or P(ham) by looking at the length of each category. With this in hand, I started breaking down the categories into word count. For both categories, I was able to get a word count for each and every word in both categories. From this point, the two categories were combined into a third category, which shows the total word count (for both classes). With all of this data, it was easy to compute conditional probabilities (such as $P(word =' hello'|spam)$ and so on. This data could then be manipulated in order to determine the category of each of the test samples.

The code used is shown below, with comments attached:

```
import numpy as np
import pandas
import sklearn
import cPickle
from collections import Counter
import math
import re

#Load in the training data
train = pandas.read_csv("spam_filter_train.txt", sep='\t',
names = ["type","Subject"])

#Group the training data by Ham or Spam
train.groupby('type')
train['length'] = map(lambda text: len(text),train['Subject'])
des =  train.groupby('type').describe()
Normal = train[train["type"]=="ham"]
Spam   = train[train["type"]=="spam"]

#Find the mean and standard deviation of the data
NM  = np.mean(Normal["length"])
NSD = np.std(Normal["length"])
SM  = np.mean(Spam["length"])
SSD = np.std(Spam["length"])

#Probability of Normal and Spam mail
PN  = float(len(Normal))/float(len(Spam)+len(Normal))
PS  = 1-PN

#Word counter for each word in Ham and Spam
Norm_Words = Counter(" ".join(Normal['Subject'].values.tolist()).split(" ")).items()
```

```
Spam_Words = Counter(" ".join(Spam['Subject'].values.tolist()).split(" ")).items()


NormT=dict()
SpamT = dict()

Norm_Num_Words = 0
Spam_Num_Words = 0
Tot_Num=0
NumN   = 0
NumS   = 0
NumN2  = 0
NumS2  = 0

#Count the total number of words in Normal and Spam

for k in range(len(Norm_Words)):
    Word = Norm_Words[k][0]
    Value = Norm_Words[k][1]
    NormT[Word] = Value
    Norm_Num_Words = Norm_Num_Words+Value

for k in range(len(Spam_Words)):
    Word = Spam_Words[k][0]
    Value = Spam_Words[k][1]
    SpamT[Word] = Value

    Spam_Num_Words = Spam_Num_Words+Value

#Add up the log of the word counts for later
Total = dict()
for k,v in SpamT.items():
    if k in NormT:
        Total[k] = (NormT[k],v)
        NyS  = np.log(v+1)
        NyN  = np.log(NormT[k]+1)
        #Tot_Num     = Tot_Num+(NyN+NyS)**2
        NumN        = NumN+NyN
        NumS        = NumS+NyS
    else:
        Total[k]=  (0,v)
        NyS  = np.log(v+1)
        NyS  = NyS
        Tot_Num     = Tot_Num+(NyS)**2
for k,v in NormT.items():
    if k not in SpamT:
        Total[k] = (v,0)
        NyN  = np.log(v+1)
```

```
        NyN=NyN
        Tot_Num     = Tot_Num+(NyN)**2


Unique_words = len(Total)
Tot_Num = np.sqrt(Tot_Num)

#Open the test data, initialize alpha
test = open("spam_filter_test.txt")
alpha = 0.000000000000001
Dec =[]
TS=0
TN=0
m= 0

for line in test:
    line = line.rstrip()
    #Extract the words from the lines
    p=re.compile('\w+')
    words=p.findall(line)
    #Count the number of words
    WordC = Counter(words)
    TN=0
    TS=0
    #Compute the Naive Bayes Values for each line
    for word in words:
        if word in Total:
            NyS1 = Total[word][1]
            NyN1 = Total[word][0]
            NyS  = np.log(NyS1+1)
            NyN  = np.log(NyN1+1)
            if (Total[word][1]==0 or Total[word][0]==0):
                NyN=NyN*np.log(2)
                NyS=NyS*np.log(2)
            else:
                NyN = NyN*np.log(1)
                NyS = NyS*np.log(1)
            NyN     = NyN/Tot_Num
            NyS     = NyS/Tot_Num
            alpha   = alpha
            NN      = (NumN)/Tot_Num
            NS      = (NumS)/Tot_Num
            #Add in some Laplace smoothing
            T1 = np.log((NyS+alpha)/float(NS+alpha))
            T2 = np.log((NyN+alpha)/float(NN+alpha))
            wN = (T1)
            wS = (T2)
            wN2     = wN/(wN+wS)
```

```
            wS2     = wS/(wN+wS)
            ti      = WordC[word]
            LN      = ti*wN2
            LS      = ti*wS2
            TS      = TS+LN
            TN      = TN+LS

    m=m+1
    #Based on Naive values, choose spam or normal
    if TN<TS:
        Dec.append(0)
    else:
        Dec.append(1)

#export the data to a file
df = pandas.DataFrame(Dec, range(1,len(Dec)+1),columns=['output'])
print df[200:220]
df.to_csv('erwaner_spamtestnew_data11.csv')
```

5 Problem 5

For this problem, we are given a training set, that, after a softmax transformation, has the form

$$p(C_k|\phi) = \frac{\exp(w_k^T\phi)}{\sum_{k=0}^{K-1}\exp(w_k^T\phi)} \tag{23}$$

and therefore the likelihood function is given by

$$p(t|w) = \prod_{n=1}^{N}\prod_{k=0}^{K-1} p(C_k|\phi(x_n))^{1(t_n=k)} \tag{24}$$

5.1 Part a

For the Error function, we aim to minimize the loss or negative log-likelihood, i.e., $E(w) = -\ln P(t|w)$, which in our problem means we aim to minimize

$$E(w) = -\ln\left(\prod_{n=1}^{N}\prod_{k=0}^{K-1}\left(\frac{\exp(w_k^T\phi)}{\sum_{k=0}^{K-1}\exp(w_k^T\phi)}\right)^{1(t_n=k)}\right) = -\ln\left(\prod_{n=1}^{N}\prod_{k=0}^{K-1}\left(\frac{\left(\exp(w_k^T\phi)\right)^{1(t_n=k)}}{\left(\sum_{k=0}^{K-1}\exp(w_k^T\phi)\right)^{1(t_n=k)}}\right)\right) \tag{25}$$

To ease notation, I have defined

$$C_{kn} = \left(\exp(w_k^T\phi)\right)^{1(t_n=k)} \tag{26}$$

$$D_{kn} = \left(\sum_{k=0}^{K-1}\exp(w_k^T\phi)\right)^{1(t_n=k)} \tag{27}$$

So simplifying (25), we can redefine it as

$$E(w) = -\ln\left(\prod_{n=1}^{N}\frac{C_{0n}}{D_{0n}}\bullet\frac{C_{1n}}{D_{1n}}\bullet\cdots\bullet\frac{C_{(K-1)n}}{D_{(K-1)n}}\right) \tag{28}$$

Further expanding, we get

$$E(w) = -\ln\left(\left(\frac{C_{01}}{D_{01}}\bullet\cdots\bullet\frac{C_{0N}}{D_{0N}}\right)\bullet\left(\frac{C_{11}}{D_{11}}\bullet\cdots\bullet\frac{C_{1N}}{D_{1N}}\right)\bullet\cdots\bullet\left(\frac{C_{(K-1)1}}{D_{(K-1)1}}\bullet\cdots\bullet\frac{C_{(K-1)N}}{D_{(K-1)N}}\right)\right) \tag{29}$$

which reduces, by the properties of logarithm, to:

$$E(w) = -\left[\ln(C_{01}) + \ln(C_{11}) + \cdots + \ln(C_{0N}) + \cdots + \ln\left(C_{(K-1)N}\right)\right] \tag{30}$$

$$+\left[\ln(D_{01}) + \ln(D_{11}) + \cdots + \ln(D_{0N}) + \cdots + \ln\left(D_{(K-1)N}\right)\right] \tag{31}$$

At this point, we are asked to find the partial derivative with respect to one vector w_j, i..e, $\bigtriangledown_{w_j}E(w)$. It is important to note that we only concern ourselves with one vector w_j, which greatly simplifies things. First, consider the C terms, where our goal becomes to find

$$\bigtriangledown_{w_j}\left\{-\left[\ln(C_{01}) + \ln(C_{11}) + \cdots + \ln(C_{0N}) + \cdots + \ln\left(C_{(K-1)N}\right)\right]\right\} \tag{32}$$

At this point, we really need concern ourselves with the C_{jn} terms, because these are the only terms that will have a non-zero derivative with respect to j (up until we consider the indicator function). Even if the indicator function was always 1, the terms that don't have a j would still have a zero derivative. Therefore, to save notation I will only consider the terms with j in them, i.e.,

$$\bigtriangledown_{w_j} \{-[ln(C_{j0}) + ln(C_{j1}) + \cdots + ln(C_{jN})]\} \tag{33}$$

Now, we must consider the indicator function. There is no way to know how many times the indicator function will be 1 or 0 for $t_n = j$. Therefore, to generalize I will assume that, for $n = \{1, \ldots, N\}$, $t_n = j$ N_j many times. This further simplifies the above to

$$\bigtriangledown_{w_j} \{-[N_j ln(C_j)]\} = \bigtriangledown_{w_j} \left\{-\left[N_j ln(\exp(w_j^T \phi))\right]\right\} = \bigtriangledown_{w_j} \left\{-N_j(w_j^T \phi)\right\} \tag{34}$$

Now we move onto the denominator, i.e.,

$$\bigtriangledown_{w_j} \left[\ln(D_{01}) + \ln(D_{11}) + \cdots + \ln(D_{0N}) + \cdots + \ln\left(D_{(K-1)N}\right)\right] \tag{35}$$

First note that each of the D terms has a w_j term in it (because each D term is a sum of w_k terms), so no terms can be discounted to have a gradient of zero. Next, it is important to note that t_n can only take 1 value for $n \in \{1, \ldots, N\}$. Therefore, for each sequence $\ln(D_{01}) + \ln(D_{11}) + \cdots + \ln(D_{0N})$, only one of these terms will have a non-zero indicator function value. And, since all of the D terms (with an indicator function of 1) are the same, we can really generalize $\ln(D_{01}) + \ln(D_{02}) + \cdots + \ln(D_{0N}) = \ln(D)$ where

$$D = \sum_{k=0}^{K-1} \exp(w_k^T \phi) \tag{36}$$

Therefore, N many of these sequences will result in

$$\bigtriangledown_{w_j} \left[N \ \ln\left(\sum_{k=0}^{K-1} \exp(w_k^T \phi)\right)\right] \tag{37}$$

and our overall partial derivative is:

$$\bigtriangledown_{w_j} \left[-N_j(w_j^T \phi) + N \ \ln\left(\sum_{k=0}^{K-1} \exp(w_k^T \phi)\right)\right] \tag{38}$$

and we aim to find where $\bigtriangledown_{w_j} E(w) = 0$, i.e., where

$$\bigtriangledown_{w_j} \left[-N_j(w_j^T \phi) + N \ \ln\left(\sum_{k=0}^{K-1} \exp(w_k^T \phi)\right)\right] = 0 \tag{39}$$

And now we can solve, i.e.,

$$-N_j \phi + N \left(\frac{\phi \exp(w_j^T \phi)}{\sum_{k=0}^{K-1} \exp(w_k^T \phi)}\right) = 0 \tag{40}$$

$$N \phi \exp(w_j^T \phi) = N_j \phi \sum_{k=0}^{K-1} \exp(w_k^T \phi) \tag{41}$$

for our purposes, we define $\hat{D}$ as D but without the $\exp(w_j^T\phi)$ term in the sum. This way, we can move the $exp(w_j^T\phi)$ from the sum to the other side of the equation. With this in hand, the solution can be shown to be

$$w_j^T\phi = \ln\left(\frac{N_j\hat{D}}{N-N_j}\right) \tag{42}$$

and w_j can be found to be any vector which satisfies the above equation.

5.2 Part b

For this section, we modify the error function by adding a weight decay term, $\sum_{k=0}^{K-1} w_k^T w_k$, such that our error function is now:

$$E^\lambda(w) = E(w) + \frac{\lambda}{2}\sum_{k=0}^{K-1} w_k^T w_k \tag{43}$$

We are asked to find the gradient of the modified error function $E^\lambda(w)$ with respect to one of the parameter vectors w_j, i.e., $\bigtriangledown_{w_j} E^\lambda(w)$. From the previous section, we know

$$\bigtriangledown_{w_j} E(w) = -N_j\phi + N\left(\frac{\phi\exp(w_j^T\phi)}{\sum_{k=0}^{K-1}\exp(w_k^T\phi)}\right) \tag{44}$$

and the partial derivative w.r.t w_j of $\frac{\lambda}{2}\sum_{k=0}^{K-1} w_k^T w_k$ is

$$\bigtriangledown_{w_j}\left(\frac{\lambda}{2}\sum_{k=0}^{K-1} w_k^T w_k\right) = \frac{\lambda}{2}(w_j + w_j) = \lambda w_j \tag{45}$$

and therefore the overall error expression is

$$\bigtriangledown_{w_j} E^\lambda(w) = -N_j\phi + N\left(\frac{\phi\exp(w_j^T\phi)}{\sum_{k=0}^{K-1}\exp(w_k^T\phi)}\right) + \lambda w_j \tag{46}$$

with a significantly more involved closed-form solution.