EECS 545 – Machine Learning - Homework #2

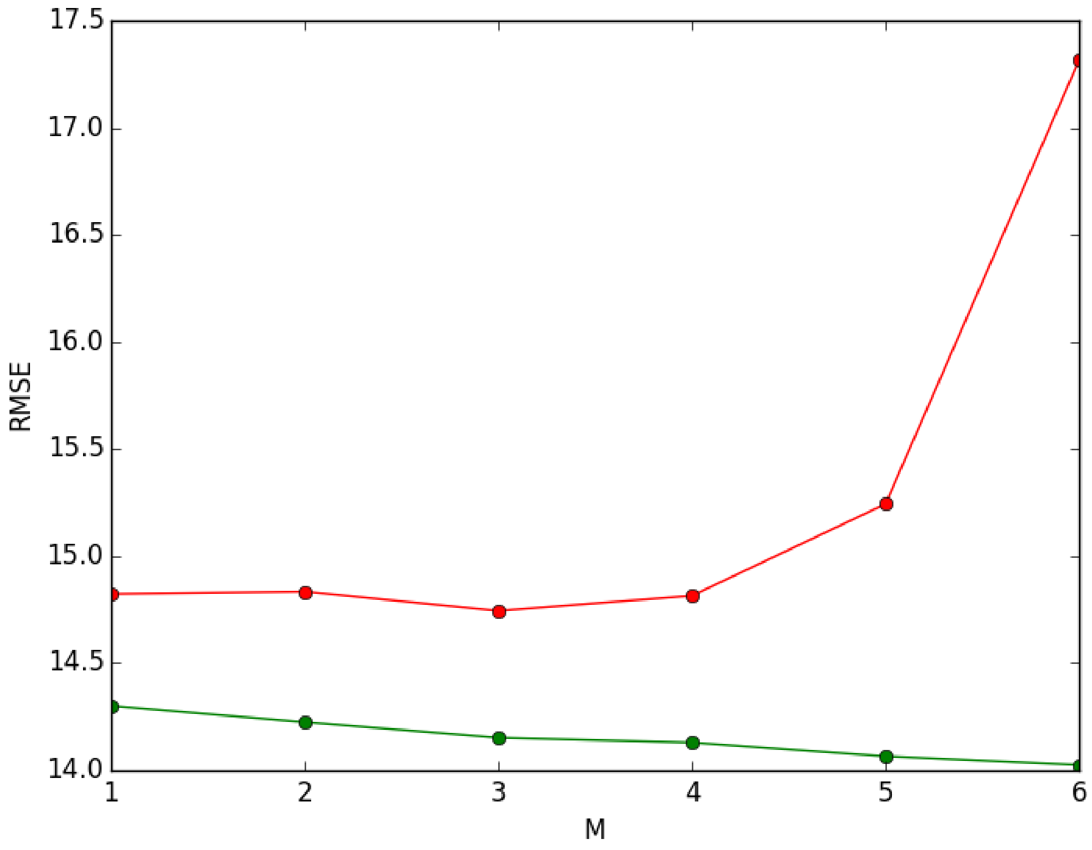
Meng Huang Due: 11:00pm 02/08/2016

Homework Policy: Working in groups is fine, but each member must submit their own writeup. Please write the members of your group on your solutions. There is no strict limit to the size of the group but we may find it a bit suspicious if there are more than 4 to a team. Questions labelled with **(Challenge)** are not strictly required, but you'll get some participation credit if you have something interesting to add, even if it's only a partial answer. **For coding problems, please append your code to your submission and report your results (values, plots, etc.) in your written solution**. You will lose points if you only include them in your code submissions. Homework will be submitted via Gradescope (https://gradescope.com/).

1) **Linear Regression (20 pts).** In this problem, you will implement linear regression (using polynomial features) to reproduce curves similar to those you have seen in lecture, exploring how various choices of model-complexity parameters affect training and test error.

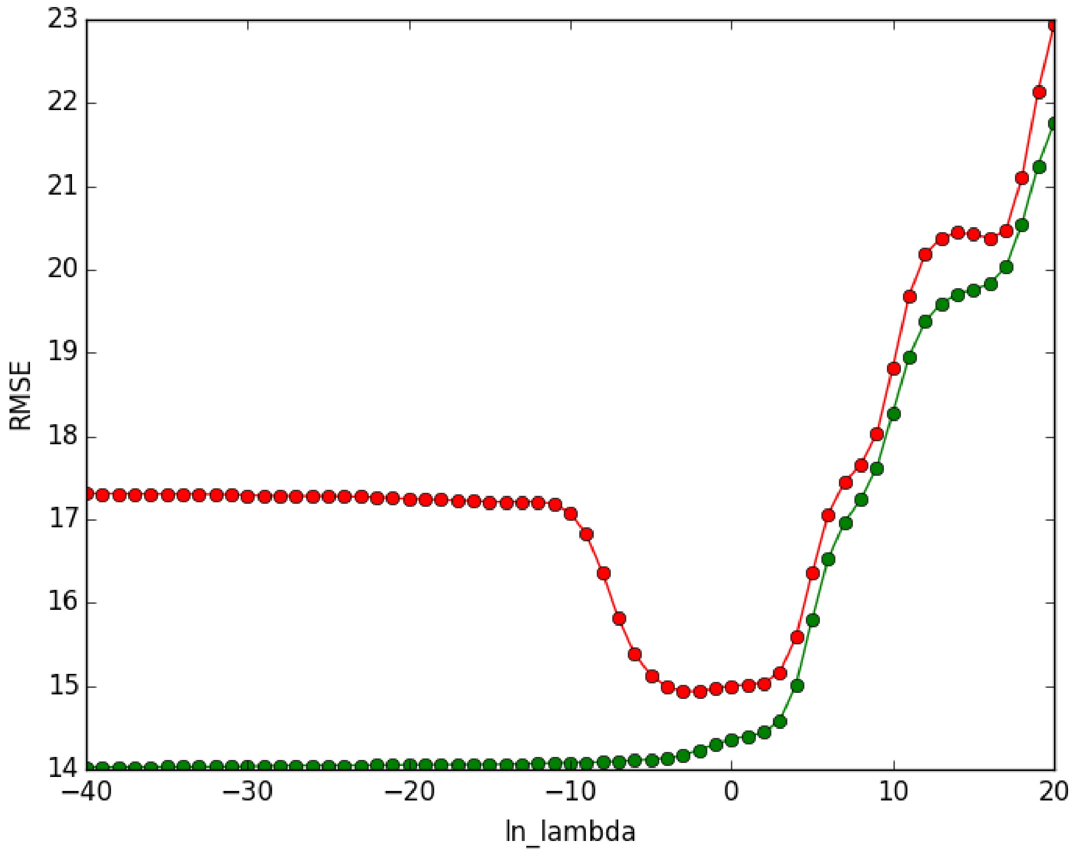
(a) The training and testing data is provided to you, named **train_graphs_f16_autopilot_cruise.csv** and **test_graphs_f16_autopilot_cruise.csv**. Every row in these csv files correspond to a single datapoint $\mathbf{x}$, $\mathbf{t}$. Columns of this graphs are named id, rolling speed, elevation speed, elevation jerk, elevation, roll, elevation acceleration, controller input. Your task is to predict controller input from rolling speed, elevation speed, elevation jerk, elevation, roll, and elevation acceleration. Note that id is a primary key, and is unique to each datapoint, so you'd want to remove it from the feature set. Ponder why this is necessary, and what would happen otherwise. Also, you might need to append an additional feature that is constant for all data points, for example a feature with a value of 1, in order to model the intercept term.

(i) Fit the data by applying the psuedo-inverse approach of linear regression using x_j^i, $i = 1, \ldots, M$ as features (try $M = 1, 2, \ldots, 6$), where x_j represents the j^{th} component of vector $\mathbf{x}$. In other words, you will need to raise every element in vector $\mathbf{x}$ to the power of i, for $i = 1, 2, \ldots, M$ to get the set of features. For example, if $\mathbf{x} = [x_1, x_2]'$ and $M = 2$, then you'd have 4 features, $x_1^1, x_1^2, x_2^1, x_2^2$ along with an additional feature 1 for the intercept term. Plot training error and test error as Root Mean Square Error (RMSE) against M, the order of the polynomial features. Name the generated plot 1.png.



Note: the red one is for test error and the green one is for train error

(ii) Fit the data by using psuedo-inverse approach of regularized linear regression. For this, you need to use all polynomial features (i.e. $M = 6$), and choose values of $\ln\lambda$ using a sweep as follows: $-40, -39, \ldots, 18, 19, 20$. Plot training error and test error as Root Mean Square Error (RMSE) against $\ln\lambda$, the regularization coefficient. Name the generated plot 2.png.

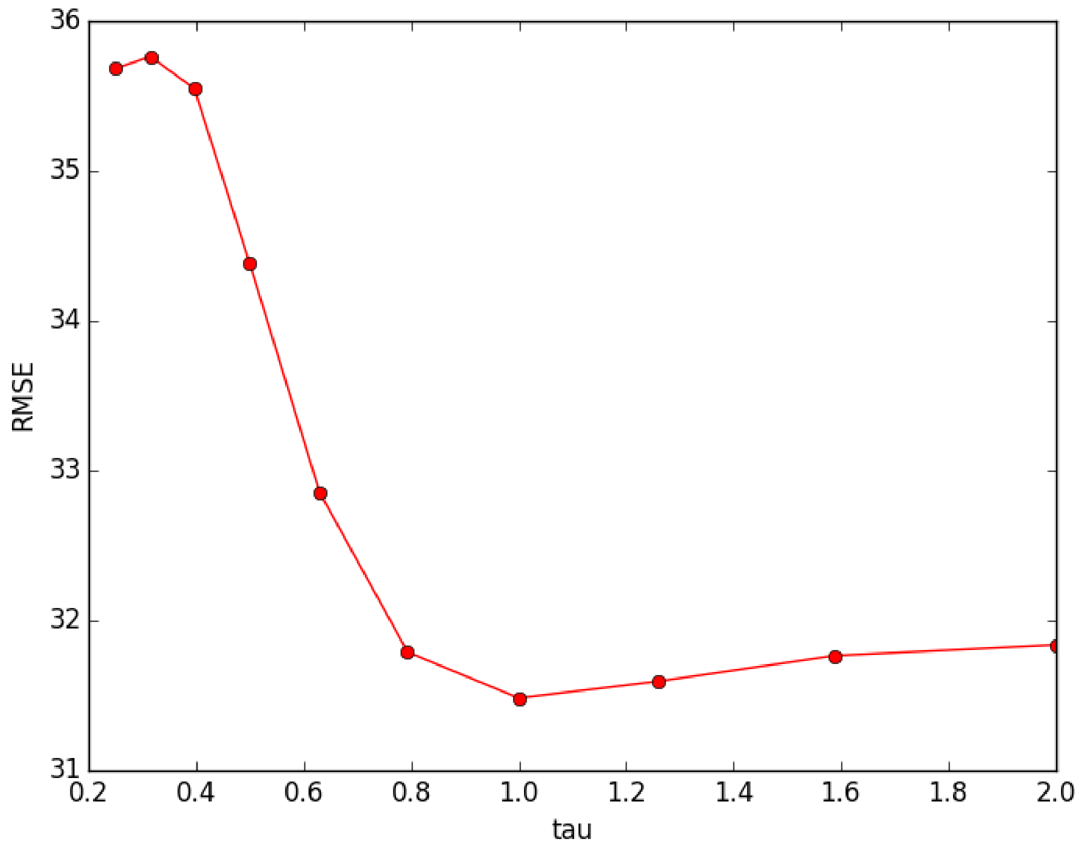


Note: the red one is for test error and the green one is for train error

(b) Because locally weighted linear regression is computationally more expensive, this problem is provided with a smaller test dataset. The training and testing data is provided to you, named **train_graphs_f16_autopilot_cruise.csv** and **test_locreg_f16_autopilot_cruise.csv**. For this part, you will not create the features by yourself, but rather use the features given in the dataset.
Fit the data by applying the psuedo-inverse approach for solving locally weighted linear regression. Use as weights the similarity between the point of interest and the example point as defined by

$$r_i(\mathbf{x}) = \exp(-\frac{||\mathbf{x}^{(i)} - \mathbf{x}||^2}{2\tau^2})$$

and choose values of τ using a geometric sweep as follows: Generate 10 evenly spaced numbers on a logarithmic scale between 2^{-2} and 2^1 (start and end values inclusive). Plot only the test error as Root Mean Square Error (RMSE) against τ, a hyperparameter that affects the performance of the algorithm. Name the generated plot 3.png. Note: You might want to think about formulating efficient matrix multiplications in your code to implement locally weighted linear regression for this problem.



Code:

```
import numpy as np
from matplotlib import pyplot as plt
from sklearn import preprocessing
import math, csv

def readData(filename):
  in_f = open(filename, 'rb')
  csv_file = csv.reader(in_f)

  X = []
  t = []
  i = 0
  for row in csv_file:
    if i == 0:
      i += 1
      continue
    X.append([float(elt) for elt in row[1:-1]])
    t.append(float(row[-1]))

  in_f.close()

  return X, t

def getFeatureMatrix(X, M):
  # create feature matrix
  feature_matrix = []
  for x in X:
    feature_vector = [1.0]
    for entry in x:
      for m in range(1, M+1):
        feature_vector.append(entry ** m)
    feature_matrix.append(feature_vector)
  return feature_matrix

def rmse(predictions, targets):
  # calculate RMSE between predictions and targets
  predictions = np.array(predictions)
  targets = np.array(targets)
  return np.sqrt(((predictions - targets) ** 2).mean())


''' Problem 1.(a)i '''
def partA():
  print 'partA() start...'
  # read training and testing data from files
  train_X, train_t = readData('train_graphs_f16_autopilot_cruise.csv')
  test_X, test_t = readData('test_graphs_f16_autopilot_cruise.csv')
```

```
# initialize parameters
list_M = range(1, 7)
K = 6

list_train_error = []
list_test_error = []

# Psuedo-inverse approach for linear regression
for M in list_M:
  # get Phi of train data
  feature_matrix = getFeatureMatrix(train_X, M)
  Phi = np.mat(feature_matrix)
  t_vector = np.mat(train_t).T

  # get w*
  left = Phi.T * Phi
  right = Phi.T * t_vector

  w = np.linalg.solve(left, right)

  # calculate train error
  predictions = Phi * w
  predictions = np.squeeze(np.asarray(predictions))

  train_error = rmse(predictions, train_t)
  list_train_error.append(train_error)

  # get Phi of test data
  feature_matrix = getFeatureMatrix(test_X, M)
  Phi = np.mat(feature_matrix)

  # calculate test error
  predictions = Phi * w
  predictions = np.squeeze(np.asarray(predictions))

  test_error = rmse(predictions, test_t)
  list_test_error.append(test_error)

# Plot
fig = plt.figure()
plt.plot(list_M, list_train_error, '-og')
plt.plot(list_M, list_test_error, '-or')
plt.xlabel('M')
plt.ylabel('RMSE')
fig.savefig('1.png')

print 'partA() end'
```

```
''' Problem 1.(a)ii '''
def partB():
  print 'partB() start...'
  # read training and testing data from files
  train_X, train_t = readData('train_graphs_f16_autopilot_cruise.csv')
  test_X, test_t = readData('test_graphs_f16_autopilot_cruise.csv')

  # initialize parameters
  list_ln_lambda = range(-40, 21)
  M = 6

  list_train_error = []
  list_test_error = []

  # Psuedo-inverse approach for regularized linear regression
  for ln_lambda in list_ln_lambda:
    # get Phi of train data
    feature_matrix = getFeatureMatrix(train_X, M)
    Phi = np.mat(feature_matrix)
    t_vector = np.mat(train_t).T

    # get w*
    left = Phi.T * Phi + np.exp(ln_lambda) * np.identity(Phi.shape[1])
    right = Phi.T * t_vector

    w = np.linalg.solve(left, right)

    # calculate train error
    predictions = Phi * w
    predictions = np.squeeze(np.asarray(predictions))

    train_error = rmse(predictions, train_t)
    list_train_error.append(train_error)

    # get Phi of test data
    feature_matrix = getFeatureMatrix(test_X, M)
    Phi = np.mat(feature_matrix)

    # calculate test error
    predictions = Phi * w
    predictions = np.squeeze(np.asarray(predictions))

    test_error = rmse(predictions, test_t)
    list_test_error.append(test_error)

  # Plot
  fig = plt.figure()
  plt.plot(list_ln_lambda, list_train_error, '-og')
  plt.plot(list_ln_lambda, list_test_error, '-or')
  plt.xlabel('ln_lambda')
  plt.ylabel('RMSE')
  fig.savefig('2.png')

  print 'partB() end'



''' Problem 1.(b) '''
def partC():
  print 'partC() start...'
  def gausssian(mu, tau, data):
    mu = np.mat(mu)
    data = np.mat(data)

    diff = data - mu
    d = float(diff * diff.T)
    return np.exp(-1.0 * d / (2 * (tau ** 2)))

  def getWeight(mu, tau, X):
    list_r = []
    for x in X:
      r = gausssian(mu, tau, x)
      list_r.append(r)
    return np.diag(list_r)

  # read data from files
  train_X, train_t = readData('train_graphs_f16_autopilot_cruise.csv')
  test_X, test_t = readData('test_locreg_f16_autopilot_cruise.csv')

  # initialize
  list_tau = np.logspace(-2, 1, num=10, base=2)
  M = 1

  # standardization
  scaler = preprocessing.StandardScaler().fit(train_X + test_X)
  train_X_std = scaler.transform(train_X)
  test_X_std = scaler.transform(test_X)

  list_test_error = []

  # Psuedo-inverse approach for regularized linear regression

  # get Phi of training data
  feature_matrix = getFeatureMatrix(train_X_std, M)
  Phi = np.mat(feature_matrix)
  t_vector = np.mat(train_t).T

  for tau in list_tau:
    print tau
    predictions = []
    for (x_test, t_test) in zip(test_X_std, test_t):
            # get the weight of x_test with each training data
            R = getWeight(x_test, tau, train_X_std)

            # get w*
            left = Phi.T * R * Phi
            right = Phi.T * R * t_vector
            w = np.linalg.solve(left, right)

            y = np.mat([1.0] + list(x_test)) * w
            predictions.append(y)

        test_error = rmse(predictions, test_t)
        print test_error
        list_test_error.append(test_error)

    # Plot
    fig = plt.figure()
    plt.plot(list_tau, list_test_error, '-or')
    plt.xlabel('tau')
    plt.ylabel('RMSE')
    fig.savefig('3.png')

    print 'partC() end'



def main():
    partA()
    partB()
    partC()

if __name__ == '__main__':
    main()
```

2) **Open Kaggle challenge (15 pts).** You can use any algorithm and design any features to perform regression on the Steel Ultimate Tensile Strength Dataset. All details for this project are included in the Kaggle webpage **https://inclass.kaggle.com/c/steel-ultimate-tensile-strength**. This problem will be graded separately based on your performance on the private leaderboard.

Code:

```
from sklearn.linear_model import RidgeCV
from sklearn.metrics import mean_squared_error
import numpy as np
from matplotlib import pyplot as plt

def readData(filename, test=False):
  # read data from files
  cols = (1,2,3,4,5,6,7,8,9)
  if test:
    cols = (1,2,3,4,5,6,7,8)
  data = np.loadtxt(filename, delimiter=",", skiprows=1, usecols=cols)
  X = data[:, :-1]
  t = data[:, -1]
  if test:
    X = data[:, :]
    t = None
  return X, t

def main():
  train_X, train_t = readData('steel_composition_train.csv')
  test_X, test_t = readData('steel_composition_test.csv', test=True)
  cols = [0, 3, 5, 6, 7]
  for j in cols:
    for i in xrange(len(train_X)):
      train_X[i][j] = np.log(train_X[i][j])

  # training data
  alphas = np.logspace(-2, 2, 50)
  ridge_cv = RidgeCV(alphas)
  ridge_cv.fit(train_X, train_t)

  for j in cols:
    for i in xrange(len(test_X)):
      test_X[i][j] = np.log(test_X[i][j])

  # predicting data
  pred_t = ridge_cv.predict(test_X)

  output_matrix = []
  for (i, t) in zip(range(1, len(pred_t)+1), pred_t):
    output_matrix.append([i, t])
  output_matrix = np.mat(output_matrix)
  np.savetxt('final_result.csv', output_matrix, fmt=('%i', '%.2f'), delimiter=',', header=
```

3) **Weighted Linear Regression (15 pts).** Consider a linear regression problem in which we want to weigh different training examples differently. Specifically, suppose we want to minimize

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

In class, we worked out what happens for the case where all the weights (r_i's) are one. In this problem, we will generalize some of those ideas to the weighted setting. In other words, we will allow the weights r_i to be different for each of the training examples.

(a) Show that $E_D(w)$ can also be written as

$$E_D(w) = (Xw - t)^\top R(Xw - t) \tag{2}$$

for an appropriate definition of X, t and R, i.e., write the derivation as part of your submitted answer. State these three quantities clearly as part of your answer (i.e., specify the form, matrix, vector, scalar, their dimensions, and how to build them from the x_i's, t_i's and r_i's.

Answer: Let

$$R = \frac{1}{2}diag\{r_1, r_2, ..., r_N\}$$

$$X = \begin{pmatrix} x_1^\top \\ x_2^\top \\ . \\ . \\ . \\ x_N^\top \end{pmatrix}$$

$$t = (t_1, t_2, ..., t_N)^\top$$

which R and X are $N \times N$ matrices and t is an N dimensional vector
Then

$$Xw - t = \begin{pmatrix} x_1^\top w - t_1 \\ x_2^\top w - t_2 \\ . \\ . \\ . \\ x_N^\top w - t_N \end{pmatrix} = \begin{pmatrix} w^\top x_1 - t_1 \\ w^\top x_2 - t_2 \\ . \\ . \\ . \\ w^\top x_N - t_N \end{pmatrix}$$

So

$$(Xw - t)^\top R(Xw - t) = \sum_{i=1}^{N} \frac{1}{2} r_i(w^\top x_i - t_i)^2 = \frac{1}{2}\sum_{i=1}^{N} r_i(w^\top x_i - t_i)^2$$

■

(b) If all the r_i's are equal to 1, we showed in class that the normal equation is

$$X^\top X w = X^\top t \tag{3}$$

and that the value of w^* that minimizes $E_D(w)$ is given by $(X^T X)^{-1} X^T t$. By finding the derivative $\nabla_w E_D(w)$ and setting that to zero, generalize the normal equation to this weighted setting, and give the new value of w^* that minimizes $E_D(w)$ in closed form as a function of X, R and t.

Answer: First, we need to find the derivative $\nabla_w E_D(w)$

$$\begin{aligned}\nabla_w E_D(w) &= \nabla_w[(Xw - t)^\top R(Xw - t)] \\ &= \nabla_w[(w^\top X^\top - t^\top)R(Xw - t)] \\ &= \nabla_w[(w^\top X^\top RXw - 2t^\top RXw + t^\top Rt)] \\ &= 2X^\top RXw - 2t^\top RX\end{aligned}$$

So, if we set $\nabla_w E_D(w)$ to 0, then

$$\begin{aligned} & 2X^\top RXw^* - 2t^\top RX = 0 \\ \Rightarrow \quad & X^\top RXw^* = X^\top Rt \\ \Rightarrow \quad & w^* = (X^\top RX)^{-1} X^\top Rt\end{aligned}$$

■

(c) Suppose we have a training set $\{(x_i, t_i); i = 1, \cdots, N\}$ of N independent examples, but in which the t_i's were observed with different variances. Specifically, suppose that

$$p(t_i|x_i; w) = \frac{1}{\sqrt{2\pi}\sigma_i} \exp\Big(-\frac{(t_i - w^T x_i)^2}{2(\sigma_i)^2}\Big). \tag{4}$$

In other words, t_i has mean $w^T x_i$ and variance $(\sigma_i)^2$ (where the σ_i's are fixed, **known** constants). Show that finding the maximum likelihood estimate of w reduces to solving a weighted linear regression problem. State clearly what the r_i's are in terms of the σ_i's.

Proof: The likelihood is

$$\prod_{i=1}^N p(t_i|x_i; w) = \prod_{i=1}^N \frac{1}{\sqrt{2\pi}\sigma_i} \exp\Big(-\frac{(t_i - w^T x_i)^2}{2(\sigma_i)^2}\Big)$$

$$\Rightarrow \log \prod_{i=1}^N p(t_i|x_i; w) = \sum_{i=1}^N \log[\frac{1}{\sqrt{2\pi}\sigma_i} \exp\Big(-\frac{(t_i - w^T x_i)^2}{2(\sigma_i)^2}\Big)]$$

$$= -N \log(\sqrt{2\pi}\sigma_i) - \sum_{i=1}^N \frac{1}{2(\sigma_i)^2}(t_i - w^T x_i)^2$$

To find the maximum likelihood estimate of w, we need to find the derivative of the log likelihood first, and set it to 0 to find the w^*

$$\nabla_w \log \prod_{i=1}^N p(t_i|x_i; w) = -\nabla_w \sum_{i=1}^N \frac{1}{2(\sigma_i)^2}(t_i - w^T x_i)^2$$

So, if we set $r_i = \frac{1}{\sigma_i^2}$, then it becomes to the form $-\nabla_w \frac{1}{2} \sum_{i=1}^N r_i(t_i - w^T x_i)^2 = 0$, which can be solved as $w^* = (X^\top R X)^{-1} X^\top R t$ by Problem (b). (use the definition of R, X, t in Problem (a)) $\square$

4) **Naive Bayes Classifier (35 pts).**

(a) Download the files **spambase.train** and **spambase.test**.
The file spambase.train contains 2000 training data and spambase.test has 2601 test data. Both datasets have 58 columns: the first 57 columns are input features, corresponding to different properties of an email, and the last column is an output label indicating spam (1) or non-spam (0). Please fit the Naive Bayes model using the training data.
As a pre-processing step, quantize each variable to one of two values, say 1 and 2, so that values below the median map to 1, and others map to 2. Please aggregate the training and test test to obtain the median value of each variable, and then use the median to quantize both data sets.

(i) Look up definitions of nominal/ordinal/interval/ratio variables. Which one(s) of them are suitable for the pre-processing described above? Look up what features are used in spambase data. Are they all suitable? Report briefly.

Answer: According to the definition of these four variables, both of them are suitable for the pre-processsing. The nominal is just "label", so it is suitable. The ordinal will indicate the level of importance and significance, which can be use to analyze the significant features in spam filter. The interval can also show the exact differences between the values. And the ratio has absolute zero, which allows for a wide range of both descriptive and inferential statistics.
The feature used in spambase data are all suitable. Each of them can describe one important information in the email.

(ii) Report the test error (misclassification percentage) of Naive Bayes classifier. As a sanity check, what would be the test error if you always predicted the same class, namely, the majority class from the training data?

Answer: According to my python code, the test error of Naive Bayes is 10.5344%, and the sanity check's test error is 45.0211% (Code is below)

(b) Open Kaggle challenge: you can design any features to perform Naive Bayes classification on the Naive Bayes Spam Filter Dataset. All details for this project are included in the Kaggle webpage **https://inclass.kaggle.com/c/naive-bayes-spam-filter**. A utility script `extract_feature.py` that performs lemmatization and stopwords removal of email texts is provided. You may adapt it for your feature extraction.
This problem will be graded separately based on your performance on the private leaderboard. Besides submitting your solutions to Kaggle, briefly describe how you extract features from email texts.

Answer: In the *extract_feature.py*, I remove all the punctuation from the original text, and do lemmatizing, stemming, removing stopwords and tf feature (Code is below)

Code for problem 4(a):

```
import math
import numpy as np

def readData(filename):
  in_f = open(filename, 'rb')

  feature_map = []
  categories = []
  for line in in_f:
    data = line.strip().split(',')
    features = [float(entry) for entry in data[:-1]]
    category = float(data[-1])
    feature_map.append(features)
    categories.append(category)

  in_f.close()
  return feature_map, categories

def getMedian(l):
  l = sorted(l)
  n = len(l)
  if n % 2:
    return l[n/2]
  else:
    return float(l[n/2 - 1] + l[n/2]) / 2.0

def preprocess(l):
  med = getMedian(l)
  result = []
  for i in xrange(len(l)):
    if l[i] > med:
      result.append(1.0)
    else:
      result.append(0.0)
  return result

def main():
  # read data from files
  train_feature_map, train_categories = readData('spambase.train')
  test_feature_map, test_categories = readData('spambase.test')

  # preprocess
  total_feature_map = train_feature_map + test_feature_map
  for j in xrange(len(total_feature_map[0])):
    l = [total_feature_map[i][j] for i in range(len(total_feature_map))]
    l = preprocess(l)
    for i in xrange(len(total_feature_map)):
      total_feature_map[i][j] = l[i]
```

```
 train_feature_map = total_feature_map[:2000]
 test_feature_map = total_feature_map[2000:]

# training step
N_spam = 0.0
N_nonspam = 0.0
for elt in train_categories:
  if elt == 1:
    N_spam += 1.0
  else:
    N_nonspam += 1.0
phi_spam = math.log(N_spam / (N_spam + N_nonspam))
phi_nonspam = math.log(N_nonspam / (N_spam + N_nonspam))


N_feature_spam_list = []
N_feature_nonspam_list = []
for j in xrange(len(train_feature_map[0])):
  N_feature_spam = 0.0
  N_feature_nonspam = 0.0
  for i in xrange(len(train_feature_map)):
    if train_feature_map[i][j] == 1:
      if train_categories[i] == 1:
        N_feature_spam += 1.0
      else:
        N_feature_nonspam += 1.0
  N_feature_spam_list.append(N_feature_spam)
  N_feature_nonspam_list.append(N_feature_nonspam)
S_spam = sum(N_feature_spam_list)
S_nonspam = sum(N_feature_nonspam_list)

mu_spam_list = [math.log(N/S_spam) for N in N_feature_spam_list]
mu_nonspam_list = [math.log(N/S_nonspam) for N in N_feature_nonspam_list]

# prediction part
predictions = []
for i in xrange(len(test_feature_map)):
  result_spam = phi_spam
  result_nonspam = phi_nonspam
  for j in xrange(len(test_feature_map[0])):
    if test_feature_map[i][j] == 1:
      result_spam += mu_spam_list[j]
      result_nonspam += mu_nonspam_list[j]

  # predict
  if result_nonspam < result_spam:
    predictions.append(1.0)
  else:
    predictions.append(0.0)

# calculate error
```

```
error = 0.0
for (y, t) in zip(predictions, test_categories):
  if y != t:
    error += 1.0
print error / float(len(test_categories))

# sanity check
error = 0.0
for y in predictions:
  if y == 1:
    error += 1.0

print error / float(len(test_categories))
```

```
Code for problem 4(b):



import pickle, os, csv
import numpy as np
from sklearn.naive_bayes import GaussianNB

with open(os.path.join('spam_filter_train.txt'), 'r') as f:
  targets = []
  for line in f:
    line = line.replace('\n', '').split('\t')[0]
    if line == 'ham':
      targets.append(0.0)
    if line == 'spam':
      targets.append(1.0)
  targets = np.array(targets)

with open(os.path.join('spam_filter_test.txt'), 'r') as f:
  orders = []
  for line in f:
    line = line.replace('\n', '').split('\t')[0]
    orders.append(line)

with open('trainFeatures.pkl', 'rb') as f:
  train_info = pickle.load(f)
with open('testFeatures.pkl', 'rb') as f:
  test_info = pickle.load(f)

train_X = train_info.toarray()
test_X = test_info.toarray()
m, n = train_X.shape
train_t = np.zeros(m)

gnb = GaussianNB()
gnb.fit(train_X, targets)
pred_t = gnb.predict(test_X)
with open('final_result.csv', 'wb') as f:
  csvwriter = csv.writer(f)
  csvwriter.writerow(['id', 'output'])
  for (number, result) in zip(orders, pred_t):
    csvwriter.writerow([number, result])
```

5) **Softmax Regression (15 pts).** In this problem, you will generalize logistic regression (for binary/2-class classification) to allow more classes (> 2) of labels (for multi-class classification).

In logistic regression, we had a training set $\{(\phi(\mathbf{x}_n),\, t_n)\}$ of N examples, where $t_n \in \{0, 1\}$, and the likelihood function is

$$p(\mathbf{t}|\mathbf{w}) = \prod_{n=1}^{N} y_n^{t_n}(1-y_n)^{1-t_n} \tag{5}$$

where $y_n = p(C_1|\phi(\mathbf{x}_n)) = \sigma(\mathbf{w}^T\phi(\mathbf{x}_n))$ and $1 - y_n = p(C_0|\phi(\mathbf{x}_n)) = 1 - \sigma(\mathbf{w}^T\phi(\mathbf{x}_n))$.

Now we generalize the logistic regression to multi-class classification, where the label t can take on K different values, rather than only two. We now have a training set $\{(\phi(\mathbf{x}_n),\, t_n)\}$ of N examples, where $t_n \in$ {0,1, ..., K-1}. We apply a softmax transformation of linear functions of the feature variables ϕ for the posterior probabilities $p(C_k|\phi)$, so that

$$p(C_k|\phi) = \frac{\exp(\mathbf{w}_k^T\phi)}{\sum_{k=0}^{K-1}\exp(\mathbf{w}_k^T\phi)}$$

where $\mathbf{w}_k$ are the parameters of the linear function for class k, and $\mathbf{w} = \{\mathbf{w}_k\}_{k=0}^{K-1}$ are the parameters for softmax regression. The likelihood function is then given by

$$p(\mathbf{t}|\mathbf{w}) = \prod_{n=1}^{N}\prod_{k=0}^{K-1} p(C_k|\phi(\mathbf{x}_n))^{\mathbf{1}(t_n=k)} \tag{6}$$

where $\mathbf{1}(.)$ is an indicator function so that $\mathbf{1}$(a true statement)=1 and $\mathbf{1}$(a false statement)=0. In the likelihood function, $\mathbf{1}(t_n = k)$ returns 1 if t_n equals k and 0 otherwise.

(a) Gradient descent for softmax regression. We can optimize the softmax regression model in the same way as logistic regression using gradient descent. Please write down the error function for softmax regression $E(\mathbf{w})$ and derive the gradient of the error function with respect to one of the parameter vectors $\mathbf{w}_j$, i.e. $\nabla_{\mathbf{w}_j}E(\mathbf{w})$. (Show your derivation and final result using the given notations. The final result should not contain any partial derivative or gradient operator.)

Answer: The error function (similar as logistic function) is:

$$E(\mathbf{w}) = -\log P(\mathbf{t}|\mathbf{w}) = -\sum_{n=1}^{N}\sum_{k=0}^{K-1}\mathbf{1}(t_n = k)\log p(C_k|\phi(\mathbf{x}_n))$$

And the derivative of this error function respect to $\mathbf{w}_j$ will be:

$$\nabla_{\mathbf{w}_j}E(\mathbf{w}) = -\sum_{n=1}^{N}\sum_{k=0}^{K-1}\nabla_{\mathbf{w}_j}[\mathbf{1}(t_n = k)\log p(C_k|\phi(\mathbf{x}_n))]$$

$$= -\sum_{n=1}^{N}\sum_{k=0}^{K-1}\mathbf{1}(t_n = k)\nabla_{\mathbf{w}_j}[\mathbf{w}_k^T\phi(\mathbf{x}_n) - \log\sum_{k=0}^{K-1}\exp(\mathbf{w}_k^T\phi(\mathbf{x_n}))]$$

$$= -\sum_{n=1}^{N}\sum_{k=0}^{K-1}\mathbf{1}(t_n = k)[\mathbf{1}(j = k) - \frac{\exp(\mathbf{w}_j^T\phi(\mathbf{x}_n))}{\sum_{k=0}^{K-1}\exp(\mathbf{w}_k^T\phi(\mathbf{x_n}))}]\phi(\mathbf{x}_n)$$

Therefore,

$$\nabla_{\mathbf{w}_j}E(\mathbf{w}) = -\sum_{n=1}^{N}\phi(\mathbf{x}_n)[\mathbf{1}(t_n = j) - P(C_j|\phi(\mathbf{x}_n))]$$

(b) Overparameterized property of softmax regression parameterization and weight decay.

Softmax regression's parameters are overparameterized, which means that for any hypothesis we might fit to the data, there are multiple parameter settings that result in the same mapping from $\phi(\mathbf{x})$ to predictions. For example, if you add a constant to every w, the softmax regression's predictions does not change. There are multiple solutions for this issue. Here we consider one simple solution: modifying the error function by adding a weight decay term, $\sum_{k=0}^{K-1} \mathbf{w}_k^T \mathbf{w}_k$:

$$E^\lambda(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\sum_{k=0}^{K-1} \mathbf{w}_k^T \mathbf{w}_k \tag{7}$$

For any parameter $\lambda > 0$, the error function $E^\lambda(\mathbf{w})$ is strickly convex, and is guaranteed to have a unique solution. Similarly, please derive the gradient of the modified error function $E^\lambda(\mathbf{w})$ with respect to one of the parameter vectors $\mathbf{w}_j$, i.e. $\nabla_{\mathbf{w}_j} E^\lambda(\mathbf{w})$. (Show your derivation and final result using the given notations. The final results should not contain any partial derivative or gradient operator.)

Answer: Since we know that $\nabla_{\mathbf{w}_j} E(\mathbf{w}) = -\sum_{n=1}^{N} \phi(\mathbf{x}_n)[\mathbf{1}(t_n = j) - P(C_j|\phi(\mathbf{x}_n))]$ from the previous problem, we can derive that:

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

$$\Rightarrow \nabla_{\mathbf{w}_j} E^\lambda(\mathbf{w}) = \nabla_{\mathbf{w}_j} E(\mathbf{w}) + \frac{\lambda}{2}\nabla_{\mathbf{w}_j}\sum_{k=0}^{K-1} \mathbf{w}_k^T \mathbf{w}_k$$

$$= -\sum_{n=1}^{N} \phi(\mathbf{x}_n)[\mathbf{1}(t_n = j) - P(C_j|\phi(\mathbf{x}_n))] + \lambda \mathbf{w}_j$$