Tasks (Classifier evaluation)

Similar to the classifier we built in the last homework, a stub has been provided that runs a logistic regressor on the beer rating data (see link above). The stub predicts whether a beer has an ABV ≥ 6.5 based on its five rating scores:

$$p(\text{positive label}) = \sigma(\theta_0 + \theta_1 \times \text{‘review/taste’} + \theta_2 \times \text{‘review/appearance’} + \theta_3 \times \text{‘review/aroma’} + \theta_4 \times \text{‘review/palate’} + \theta_5 \times \text{‘review/overall’})$$

The stub runs logistic regression with a hyperparameter $\lambda = 1.0$. We will use this stub to further improve and evaluate our classifier.

1. The code currently does not perform any train/test splits. Split the data into training, validation, and test sets, via 1/3, 1/3, 1/3 splits. Use the first third, second third, and last third of the data (respectively). After training on the training set, report the accuracy of the classifier on the validation and test sets (1 mark).

Solution:

(1) The accuracy of the classifier on the validation set is 0.90027601104;

(2) The accuracy of the classifier on the test set is 0.577813774898.

Code:

```
import numpy
from urllib.request import urlopen
import scipy.optimize
import random
from math import exp
from math import log

def parseData(fname):
  for l in urlopen(fname):
    yield eval(l)

print("Reading data...")
data = list(parseData("http://jmcauley.ucsd.edu/cse190/data/beer/beer_50000.json"))
print("done")

def feature(datum):
  feat = [1, datum['review/taste'], datum['review/appearance'], datum['review/aroma'], datum['review/palate'], datum['review/overall']]
  return feat

X = [feature(d) for d in data]
y = [d['beer/ABV'] >= 6.5 for d in data]

def inner(x,y):
  return sum([x[i]*y[i] for i in range(len(x))])

def sigmoid(x):
  return 1.0 / (1 + exp(-x))
```

```
##################################################
# Logistic regression by gradient ascent         #
##################################################

# NEGATIVE Log-likelihood
def f(theta, X, y, lam):
    loglikelihood = 0
    for i in range(len(X)):
        logit = inner(X[i], theta)
        loglikelihood -= log(1 + exp(-logit))
        if not y[i]:
            loglikelihood -= logit
    for k in range(len(theta)):
        loglikelihood -= lam * theta[k]*theta[k]
    # for debugging
    # print("ll =" + str(loglikelihood))
    return -loglikelihood

# NEGATIVE Derivative of log-likelihood
def fprime(theta, X, y, lam):
    dl = [0]*len(theta)
    for i in range(len(X)):
        logit = inner(X[i], theta)
        for k in range(len(theta)):
            dl[k] += X[i][k] * (1 - sigmoid(logit))
            if not y[i]:
                dl[k] -= X[i][k]
    for k in range(len(theta)):
        dl[k] -= lam*2*theta[k]
    return numpy.array([-x for x in dl])

X_train = X[:int(len(X)/3)]
y_train = y[:int(len(X)/3)]
X_validate = X[int(len(X)/3):2*int(len(X)/3)]
y_validate = y[int(len(X)/3):2*int(len(X)/3)]
X_test = X[2*int(len(X)/3):]
y_test = y[2*int(len(X)/3):]

##################################################
# Train                                          #
##################################################

def train(lam):
    theta,_,_ = scipy.optimize.fmin_l_bfgs_b(f, [0]*len(X[0]), fprime, pgtol = 10, args = (X_train, y_train, lam))
    return theta

###################################################
# Predict                                         #
###################################################

def performance_test(theta):
    scores = [inner(theta,x) for x in X_test]
    predictions = [s > 0 for s in scores]
    correct = [(a==b) for (a,b) in zip(predictions,y_test)]
    acc = sum(correct) * 1.0 / len(correct)
    return acc

def performance_validate(theta):
    scores = [inner(theta,x) for x in X_validate]
    predictions = [s > 0 for s in scores]
    correct = [(a==b) for (a,b) in zip(predictions,y_validate)]
    acc = sum(correct) * 1.0 / len(correct)
    return acc
###################################################
# Validation pipeline                             #
###################################################

lam = 1.0

theta = train(lam)
acc_test = performance_test(theta)
print("lambda = " + str(lam) + ":\taccuracy=" + str(acc_test))
acc_validate = performance_validate(theta)
print("lambda = " + str(lam) + ":\taccuracy=" + str(acc_validate))
```

2. Let's come up with a more accurate classifier[1] based on a few common words in the review. Build a feature vector to implement a classifier of the form

$$p(\text{positive label}) = \sigma(\theta_0 + \theta_1 \times \#\text{'lactic'} + \theta_2 \times \#\text{'tart'}\ldots),$$

where each feature corresponds to the number of times a particular word appears. Base your feature on the following 10 words: "lactic," "tart," "sour," "citric," "sweet," "acid," "hop," "fruit," "salt," "spicy." **Convert the reviews to lowercase before counting.**

Solution:
We can use regular expression to remove punctuations from the original string. After that, we can split the whole long string into separate words to compare target words we select.
The code for building this feature vector is shown below:

```
import re

def feature(datum):
   rev = datum['review/text'].lower()
   rev = rev.replace('\t', '')
   rev = re.sub(r'[^\w\s]','',rev)
   # table = str.maketrans({key: None for key in string.punctuation})
   # rev = rev.strip(string.punctuation)
   target = ['lactic','tart','sour','citric','sweet','acid','hop','fruit','salt','spicy']
   feat = [0] * 10
   for word in rev.split():
      #      word = word.translate(table)
         for i in range(10):
               if word == target[i]:
                     feat[i] += 1;
   feat.insert(0,1)
   return feat
```

3. Report the number of true positives, true negatives, false positives, false negatives, and the *Balanced Error Rate* of the classifier on the test set (1 mark).

Solution:

(1) The number of true positives of the classifier on the test set is 5839;
(2) The number of true negatives of the classifier on the test set is 201;
(3) The number of false positives of the classifier on the test set is 10554;
(4) The number of false negatives of the classifier on the test set is 74;
(5) The balanced error rate of the classifier on the test set is 0.49691290801701377.

Code:

```
import numpy
from urllib.request import urlopen
import scipy.optimize
import random
from math import exp
from math import log
import string
import re

def parseData(fname):
        with open(fname,'r') as fp:
                for l in fp.readlines():
                        yield eval(l)

print("Reading data...")
data = list(parseData("beer_50000.json"))
print("done")



def feature(datum):
    rev = datum['review/text'].lower()
    rev = rev.replace('\t', '')
    rev = re.sub(r'[^\w\s]','',rev)
    # table = str.maketrans({key: None for key in string.punctuation})
    target = ['lactic','tart','sour','citric','sweet','acid','hop','fruit','salt','spicy']
    feat = [0] * 10
    for word in rev.split():
        #       word = word.translate(table)
            for i in range(10):
                    if word == target[i]:
                            feat[i] += 1;
    feat.insert(0,1)
    return feat
```

```
X = [feature(d) for d in data]
y = [d['beer/ABV'] >= 6.5 for d in data]

def inner(x,y):
   return sum([x[i]*y[i] for i in range(len(x))])

def sigmoid(x):
   return 1.0 / (1 + exp(-x))

##################################################
# Logistic regression by gradient ascent         #
##################################################

# NEGATIVE Log-likelihood
def f(theta, X, y, lam):
   t = sum(y) #the number of positive samples
   f = len(y)-t #the number of negative samples
   loglikelihood = 0
   for i in range(len(X)):
      logit = inner(X[i], theta)
      loglikelihood -= log(1 + exp(-logit))
      if not y[i]:
         loglikelihood -= logit
   for k in range(len(theta)):
      loglikelihood -= lam * theta[k]*theta[k]
   # for debugging
   # print("ll =" + str(loglikelihood))
   return -loglikelihood

# NEGATIVE Derivative of log-likelihood
def fprime(theta, X, y, lam):
   dl = [0]*len(theta)
   for i in range(len(X)):
      logit = inner(X[i], theta)
      for k in range(len(theta)):
         dl[k] += X[i][k] * (1 - sigmoid(logit))
         if not y[i]:
            dl[k] -= X[i][k]
   for k in range(len(theta)):
      dl[k] -= lam*2*theta[k]
   return numpy.array([-x for x in dl])

X_train = X[:int(len(X)/3)]
y_train = y[:int(len(X)/3)]
```

```
X_validate = X[int(len(X)/3):2*int(len(X)/3)]
y_validate = y[int(len(X)/3):2*int(len(X)/3)]
X_test = X[2*int(len(X)/3):]
y_test = y[2*int(len(X)/3):]


###################################################
# Train                                            #
###################################################

def train(lam):
    theta,_,_ = scipy.optimize.fmin_l_bfgs_b(f, [0]*len(X[0]), fprime, pgtol = 10, args = (X_train,
y_train, lam))
    return theta

###################################################
# Predict                                          #
###################################################

def performance(theta):
    scores = [inner(theta,x) for x in X_test]
    predictions = [s > 0 for s in scores]
    correct = [(a==b) for (a,b) in zip(predictions,y_test)]
    TP = 0
    TN = 0
    FP = 0
    FN = 0
    for (a,b) in zip(predictions, y_test):
        if a==True:
            if b == True:
                TP += 1
            else:
                FP += 1
        else:
            if b == True:
                FN += 1
            else:
                TN += 1
    print("# of true positive is",TP)
    print("# of true negative is",TN)
    print("# of false positive is",FP)
    print("# of false negative is",FN)
    BER = (FP/(FP+TN)+FN/(FN+TP))/2.0
    print("Balanced Error Rate is", BER)
    acc = sum(correct) * 1.0 / len(correct)
    return acc

##################################################
# Validation pipeline                            #
##################################################

lam = 1.0

theta = train(lam)
print("theta is equal to", theta)
acc = performance(theta)
print("lambda = " + str(lam) + ":\taccuracy=" + str(acc))
```

4. (**Hard**) Our classifier is possibly less effective than it could be due to the issue of *class imbalance* (i.e., an uneven number of the datapoints have a positive label). Show how you would adjust the gradient ascent code provided such that the classifier would be approximately 'balanced' between the positive and negative classes. Report the Balanced Error Rate (on the train/validation/test sets) for the new classifier (1 mark).

Solution:
Due to the class imbalance, we've got a model that seems to always predict "True". One way of tackling this issue is to assign weights to different classes of samples. In our case, our model seems to predict most of the samples to be True, yet we want the model to predict more negative samples. The setting of weights is the following:

$$w_j = \frac{n}{kn_j}$$

Where n represents the total number of observations in the training set, k represents the number of different classes, here is 2, and n_j represents the number of observations in the class j.
Hence, the crucial code for this problem is the following:

```
# calculate weights
train_total = len(y_train)
positive_total = sum(y_train)
negative_total = train_total - positive_total
weight_pos = train_total/(2.0*positive_total)
weight_neg = train_total/(2.0*negative_total)

# two functions that are utilized in training
# NEGATIVE Log-likelihood
def f(theta, X, y, lam):
    t = sum(y) #the number of positive samples
    f = len(y)-t #the number of negative samples
    loglikelihood = 0
    for i in range(len(X)):
        logit = inner(X[i], theta)
        if not y[i]:
            loglikelihood -= weight_neg * logit
            loglikelihood -= weight_neg * log(1 + exp(-logit))
        else:
            loglikelihood -= weight_pos * log(1 + exp(-logit))
    for k in range(len(theta)):
        loglikelihood -= lam * theta[k]*theta[k]
    # for debugging
    # print("ll =" + str(loglikelihood))
    return -loglikelihood

# NEGATIVE Derivative of log-likelihood
def fprime(theta, X, y, lam):
```

```
dl = [0]*len(theta)
for i in range(len(X)):
    logit = inner(X[i], theta)
    for k in range(len(theta)):
        if not y[i]:
            dl[k] -= weight_neg * X[i][k]
            dl[k] += weight_neg * X[i][k] * (1 - sigmoid(logit))
        else:
            dl[k] += weight_pos * X[i][k] * (1 - sigmoid(logit))
for k in range(len(theta)):
    dl[k] -= lam*2*theta[k]
return numpy.array([-x for x in dl])
```

5. Implement a training/validation/test pipeline so that you can select the best model based on its performance on the *validation* set. Try models with $\lambda \in \{0, 0.01, 0.1, 1, 100\}$. Report the performance on the training/validation/test sets for the best value of λ (1 mark).

Solution:

Using the question 4's model, we continue to solve this problem. By trying different λ, we get the following results:

(1) $\lambda = 0,\ 0.01,\ 0.1,\ 1$

Table 1. Results on the validation set for $\lambda = 0,\ 0.01,\ 0.1,\ 1$

		Label	
		Positive	Negative
Prediction	Positive	7212	213
	Negative	8696	545

Accuracy is 0.465438617545

Balanced Error Rate is 0.41382291845658714

(2) $\lambda = 100$

Table 2. Results on the validation set for $\lambda = 100$

		Label	
		Positive	Negative
Prediction	Positive	7230	213
	Negative	8678	545

Accuracy is 0.466518660746

Balanced Error Rate is 0.4132571653763759

Therefore, the best value of λ is 100. Below are the results on the training and test set when $\lambda = 100$.

Table 3. Results on the training set for $\lambda = 100$

		Label	
		Positive	Negative
Prediction	Positive	4343	2440
	Negative	5005	4878

Accuracy is 0.553282131285

Balanced Error Rate is 0.4344165382326426

Table 4. Results on the test set for $\lambda = 100$

		Label	
		Positive	Negative
Prediction	Positive	2731	3762
	Negative	3182	6993

Accuracy is 0.583393328534

Balanced Error Rate is 0.4439635524024435

Tasks (Dimensionality Reduction)

Next, we'll run dimensionality reduction on the same data, using the word features from the previous question (you can drop the constant feature). Specifically we'll try to find the principal components of our 10 word features. For this question, use the *training* set constructed from the initial 1/3, 1/3, 1/3 splits of the data.

6. Find and report the PCA components (i.e., the transform matrix) using the week 3 code (1 mark).

Solution:

By building PCA model, we can get the transform matrix when n_components = 10:

```
[[ -6.41094839e-04   3.57933750e-03  -9.55357720e-03   9.88469685e-03
    7.83224305e-01  -1.44078493e-04   6.16429904e-01   7.33895056e-02
    6.69481963e-05   3.13945194e-02]
 [ -1.55761299e-03  -8.28525871e-03  -1.37448847e-02   1.28842459e-02
   -6.18350889e-01   3.50220592e-04   7.85531610e-01  -6.54257578e-03
   -1.07095007e-03   1.05921905e-02]
 [  4.20604956e-03   4.45053780e-02   8.92522797e-02   4.56903356e-03
   -6.24643838e-02  -4.63132532e-04  -3.94173000e-02   9.91634076e-01
    2.56312546e-04   3.49398378e-02]
 [ -5.11665982e-04   2.02832948e-02  -1.79748769e-02   2.04192948e-02
   -1.60396745e-02  -1.10735096e-04  -2.68242434e-02  -3.66351016e-02
    2.93862320e-03   9.98258831e-01]
 [  2.73913731e-02   2.30041580e-01   9.67622271e-01   4.00173472e-03
    3.04288876e-03   1.00576052e-02   2.07982312e-02  -9.68700289e-02
   -3.65786770e-04   9.73619404e-03]
 [  3.36508168e-02   9.70940156e-01  -2.33925001e-01   1.25897665e-02
   -5.67453542e-03   1.04746085e-02   1.69171946e-03  -2.21262307e-02
    1.41740638e-03  -2.50413365e-02]
 [  7.83324569e-03  -1.41173074e-02  -9.09050548e-04   9.99499037e-01
    8.98577835e-04   7.24050734e-03  -1.55896521e-02  -3.75315337e-03
    8.70814095e-04  -2.07141410e-02]
 [  9.98213209e-01  -3.95517101e-02  -1.93327379e-02  -8.64189247e-03
   -9.19671968e-05   3.93825765e-02   1.25208318e-03  -6.57068581e-04
   -1.80822316e-03   1.16168520e-03]
 [ -4.00379110e-02  -1.08014721e-02  -6.47956922e-03  -7.06975448e-03
    3.27634509e-04   9.99083509e-01  -3.74997264e-04   1.72938827e-03
   -4.13097154e-03   4.08530094e-04]
 [  1.59392468e-03  -1.47606534e-03   6.40452023e-04  -9.79645608e-04
   -6.41963209e-04   4.18182307e-03   9.08441279e-04  -1.53281787e-04
    9.99983456e-01  -2.87238619e-03]]
```

Code:

```
import numpy
from urllib.request import urlopen
```

```
import scipy.optimize
import random
from math import exp
from math import log
from sklearn.decomposition import PCA
import re

def parseData(fname):
  for l in urlopen(fname):
    yield eval(l)

print("Reading data...")
data = list(parseData("http://jmcauley.ucsd.edu/cse190/data/beer/beer_50000.json"))
print("done")

def feature(datum):
  rev = datum['review/text'].lower()
  rev = rev.replace('\t', '')
  rev = re.sub(r'[^\w\s]','',rev)
  # table = str.maketrans({key: None for key in string.punctuation})
  target = ['lactic','tart','sour','citric','sweet','acid','hop','fruit','salt','spicy']
  feat = [0] * 10
  for word in rev.split():
    #     word = word.translate(table)
      for i in range(10):
          if word == target[i]:
              feat[i] += 1;
  return feat

X = [feature(d) for d in data]
pca = PCA(n_components = 10)
pca.fit(X_train)
eigen_matrix = pca.components_
print(pca.components_)
```

7. Suppose we want to compress the data using just two PCA dimensions. How large is the reconstruction error when doing so (1 mark)?[2]

Solution:
The reconstruction error can be calculated by

$$\sum_{y}\sum_{j=K+1}^{M}(y_i-\bar{y_i})^2$$

Which is equal to the variance in the discarded dimensions.
Here we only take first two dimensions, so K = 2. We can utilize the built-in function pca.explained_variance_ to acquire the average variances on each dimension.
The average reconstruction error is 0.461985226861.

Code:
```
import numpy
from urllib.request import urlopen
import scipy.optimize
import random
from math import exp
from math import log
from sklearn.decomposition import PCA
import re

def parseData(fname):
  for l in urlopen(fname):
    yield eval(l)

print("Reading data...")
data = list(parseData("http://jmcauley.ucsd.edu/cse190/data/beer/beer_50000.json"))
print("done")

def feature(datum):
  rev = datum['review/text'].lower()
  rev = rev.replace('\t', '')
  rev = re.sub(r'[^\w\s]','',rev)
  # table = str.maketrans({key: None for key in string.punctuation})
  target = ['lactic','tart','sour','citric','sweet','acid','hop','fruit','salt','spicy']
  feat = [0] * 10
  for word in rev.split():
    #     word = word.translate(table)
      for i in range(10):
          if word == target[i]:
              feat[i] += 1;
  return feat
```

```
X = [feature(d) for d in data]
y = ["American IPA" in b['beer/style'] for b in data]

X_train = X[:int(len(X)/3)]
y_train = y[:int(len(X)/3)]
X_validate = X[int(len(X)/3):2*int(len(X)/3)]
y_validate = y[int(len(X)/3):2*int(len(X)/3)]
X_test = X[2*int(len(X)/3):]
y_test = y[2*int(len(X)/3):]

pca = PCA(n_components = 10)
pca.fit(X_train)
eigen_matrix = pca.components_
print(len(X_train) * numpy.sum(pca.explained_variance_[2:]))
```

8. Looking at the first two dimensions of our data in the PCA basis is an effective way to 'summarize' the data via a 2-d plot. Using a plotting program of your choice, make a 2-d scatterplot showing the difference between 'American IPA' style beers versus all other styles (e.g. plot American IPAs in red and other styles in blue) (1 mark).

Solution:

Using the built-in library function, we can plot the figure shown in fig 1.

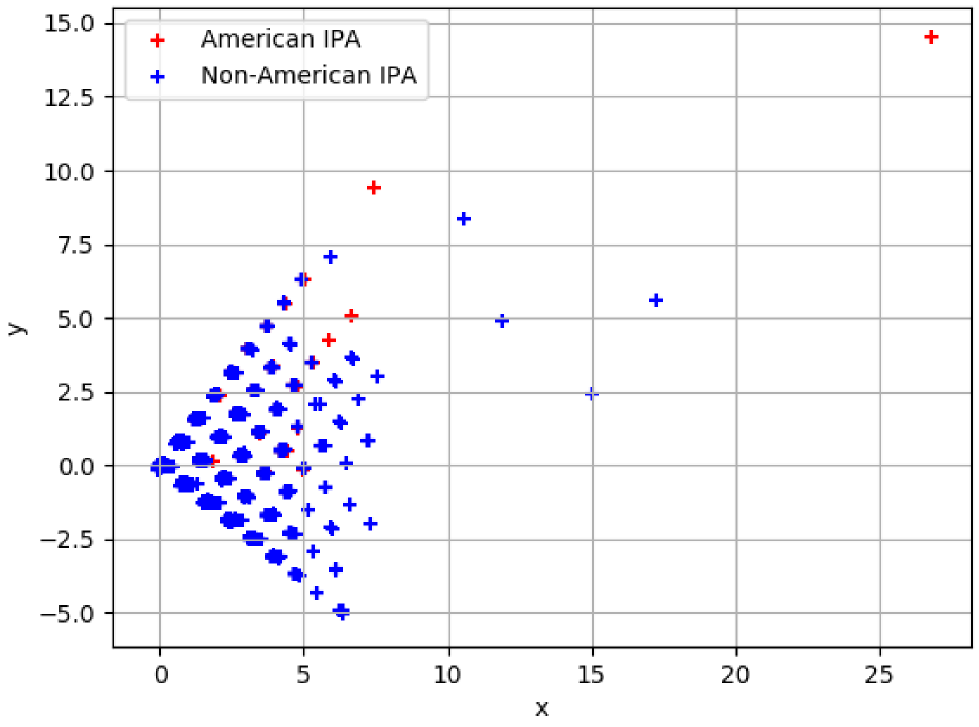

Fig. 1 data in 2-d using PCA basis

Code:

```
import numpy
from urllib.request import urlopen
import scipy.optimize
import random
from math import exp
from math import log
from sklearn.decomposition import PCA
import re

def parseData(fname):
    for l in urlopen(fname):
        yield eval(l)

print("Reading data...")
data = list(parseData("http://jmcauley.ucsd.edu/cse190/data/beer/beer_50000.json"))
print("done")

def feature(datum):
    rev = datum['review/text'].lower()
    rev = rev.replace('\t', '')
    rev = re.sub(r'[^\w\s]','',rev)
    # table = str.maketrans({key: None for key in string.punctuation})
    target = ['lactic','tart','sour','citric','sweet','acid','hop','fruit','salt','spicy']
    feat = [0] * 10
    for word in rev.split():
    #     word = word.translate(table)
        for i in range(10):
            if word == target[i]:
                feat[i] += 1;
    return feat

X = [feature(d) for d in data]
y = ["American IPA" in b['beer/style'] for b in data]
X_train = X[:int(len(X)/3)]
y_train = y[:int(len(X)/3)]
X_validate = X[int(len(X)/3):2*int(len(X)/3)]
y_validate = y[int(len(X)/3):2*int(len(X)/3)]
X_test = X[2*int(len(X)/3):]
y_test = y[2*int(len(X)/3):]

pca = PCA(n_components = 10)
pca.fit(X_train)
eigen_matrix = pca.components_
print(pca.components_)

matrix_w = numpy.hstack((eigen_matrix[0].reshape(10,1),
                         eigen_matrix[1].reshape(10,1)))
Y = numpy.dot(X, matrix_w)
is_American_IPA = []
not_American_IPA = []
for i in range(len(y)):
    if y[i]==True:
        is_American_IPA.append(Y[i])
    else:
        not_American_IPA.append(Y[i])
x_list_1 = [r[0] for r in is_American_IPA]
y_list_1 = [r[1] for r in is_American_IPA]
x_list_2 = [r[0] for r in not_American_IPA]
y_list_2 = [r[1] for r in not_American_IPA]
# numpy.savetxt('angle1.txt',Y,fmt='%f',delimiter=' ',newline='\r\n')
import matplotlib.pyplot as plt
fig = plt.figure(0)
```

```
plt.xlabel('x')
plt.ylabel('y')
plt.title('requests')
plt.scatter(x_list_1, y_list_1, c='red', alpha=1, marker='+', label='American IPA')
plt.scatter(x_list_2, y_list_2, c='blue', alpha=1, marker='+', label='Non-American IPA')
plt.grid(True)
plt.legend(loc='best')
plt.show()
```