Regression

Task 1

```
import numpy as np
def parseData(fname):
  for l in open(fname):
    yield eval(l)

print "Reading data..."
data = list(parseData("beer.json"))

# Returns two dictionaries, with the average rating for each style,
# and #reviews for each style
def get_average_and_review_count(data):
    counts = dict()
    average = dict()

    for data_point in data:
        style = data_point['beer/style']
        review = data_point['review/taste']
        n = counts.get(style, 0)
        old_average = average.get(style, 0)

        counts[style] = n + 1
        average[style] = (old_average * n + review) / (n+1)
    return average, counts

average, counts = get_average_and_review_count(data)
print "\nThe most reviewed styles"
[[style, counts[style]] for style in counts.keys() if counts[style] >= 700]
```

```
Reading data...

The most reviewed styles:
[['Fruit / Vegetable Beer', 1355],
 ['American Pale Ale (APA)', 2288],
 ['Euro Pale Lager', 701],
 ['American Porter', 2230],
 ['Doppelbock', 873],
 ['American Barleywine', 825],
 ['English Pale Ale', 1324],
 ['Rauchbier', 1938],
 ['American IPA', 4113],
 ['American Double / Imperial IPA', 3886],
 ['Russian Imperial Stout', 2695],
 ['American Double / Imperial Stout', 5964],
 ['Scotch Ale / Wee Heavy', 2776],
 ['Old Ale', 1052],
 ['Czech Pilsener', 1501],
 ['Rye Beer', 1798]]
```

```
print "\nAverage reviews for each style, where average is >= 4"
[[style, average[style]] for style in average.keys() if average[style] >= 4]
```

```
Average reviews for each style, where average is >= 4
[['Belgian Strong Pale Ale', 4.05617088607595],
 ['American Porter', 4.081838565022416],
 ['Wheatwine', 4.186813186813188],
 ['American Barleywine', 4.064242424242424],
 ['Rauchbier', 4.067853457172355],
 ['Baltic Porter', 4.213035019455248],
 ['American IPA', 4.000850960369554],
 ['English Barleywine', 4.360902255639096],
 ['American Double / Imperial IPA', 4.033324755532658],
 ['American Wild Ale', 4.188775510204083],
 ['Russian Imperial Stout', 4.300371057513916],
 ['American Double / Imperial Stout', 4.479963112005356],
 ['Scotch Ale / Wee Heavy', 4.08339337175794],
 ['Old Ale', 4.096007604562735],
 ['Rye Beer', 4.213570634037825]]
```

Task 2

```python
from scipy import optimize

def feature(d):
    feat = [1]
    if d['beer/style'] == 'American IPA':
        feat.append(1)
    else:
        feat.append(0)
    return feat

def f(theta, X, y, lam):
    theta = np.matrix(theta).T
    X = np.matrix(X)
    y = np.matrix(y).T

    diff = X*theta - y
    diffSq = diff.T * diff
    diffSqReg = diffSq / len(X) + lam*(theta.T*theta)
    res = diffSqReg.flatten().tolist()

    return res

# Derivate of f
def fprime(theta, X, y, lam):
    theta = np.matrix(theta).T
    X = np.matrix(X)
    y = np.matrix(y).T

    diff = X*theta - y
    res = 2*X.T*diff / len(X) + 2*lam*theta
    res = np.array(res.flatten().tolist()[0])

    return res
# Extract wanted features for X
X = [feature(d) for d in data]
# We are calculating the review/taste value
y = [d['review/taste'] for d in data]
theta = [0,0]
# Running gradient descent on the data
res = optimize.fmin_l_bfgs_b(f, [0,0], fprime, args = (X, y, 0.1))
print "\nTheta:"
print res[0]
```

Answer

```
Theta:
[ 3.55048115  0.20326687]
```

$\theta_0 = 3.55048115, \theta_1 = 0.20326687$

θ_0 represents the average for a beer that is not an American IPA

θ_1 represents the extra rating a American IPA usually has compared to the average rating.

Task 3

```
# MSE error
def square_error(theta,X,y):
    theta = np.matrix(theta).T
    X = np.matrix(X)
    y = np.matrix(y).T

    diff = X*theta - y
    diffSq = diff.T * diff

    return diffSq / len(X)

training_data = data[0:n]
testing_data = data[n::]

# Extracting wanted features for X
X_train = [feature(d) for d in training_data]
X_test = [feature(d) for d in testing_data]

# Extracting feature we are calculating
y_train = [d['review/taste'] for d in training_data]
y_test = [d['review/taste'] for d in testing_data]

theta = [0, 0]

# Gradient descent on training data
theta = optimize.fmin_l_bfgs_b(f, theta, fprime, args = (X_train, y_train, 0.1))[0]
print"Theta"
print theta

print "\nMSE Training data"
print square_error(theta, X_train, y_train)

print "\nMSE Testing data"
print square_error(theta, X_test, y_test)
```

Answer

```
Theta
[ 3.53845571  0.19558705]

MSE Training data
[[ 0.68485066]]

MSE Testing data
[[ 0.61342104]]
```

Task 4

```
# We get the amount of reviews for each style based on our training data
review_count = get_average_and_review_count(training_data)[1]
# Filter out each style where #reviews >= 50
styles = sorted(list(set([d['beer/style'] for d in data if review_count[d['beer/styl

theta = [0 for x in range(len(styles)+1)]

# Extracting features for X
def feature_all(d, styles):
    feat = [1]
    for style in styles:
        if d['beer/style'] == style:
            feat.append(1)
        else:
            feat.append(0)
    return feat

# Training and test data for X
X_train = [feature_all(d, styles) for d in training_data]
X_test = [feature_all(d,styles) for d in testing_data]

# Gradient descent
theta = optimize.fmin_l_bfgs_b(f, theta, fprime, args=(X_train, y_train, 0.1))[0]

print "Theta values"
print theta

print "\nMSE Error Training"
print square_error(theta, X_train, y_train)

print "\nMSE Error Testing"
print square_error(theta, X_test, y_test)
```

Answer

```
Theta values
[  3.31586548e+00   4.97476833e-03  -2.38612413e-02   2.51161620e-02
   1.65662022e-01   1.25274089e-02  -1.22513767e-02   4.49887739e-02
   2.63189656e-01   6.66707762e-01   2.98763600e-01   9.72161032e-02
   2.77190596e-03   1.43582944e-01   8.41279990e-02   5.49112449e-03
   3.29193380e-02   1.32546873e-03   2.77272013e-02   2.29714851e-02
   2.13864740e-02   1.50516596e-01   2.91581529e-02  -1.35037453e-02
  -1.21042949e-04  -5.40085121e-03   1.55368384e-02   3.66970067e-02
   1.64031407e-02   1.90976485e-02   9.90036725e-03  -1.29586827e-03
   4.44001761e-02   7.88522603e-03   2.38102761e-02   8.25645121e-03
  -6.41941592e-02  -4.16144065e-02   2.75815410e-02   1.07770525e-01
   3.07321525e-02  -1.07558062e-02   1.24299751e-02  -1.51895491e-01
   2.39937542e-03   9.03973362e-03   6.76760992e-02   1.43895635e-02
   5.10720829e-01   3.03164693e-02   1.73990850e-02   4.02272347e-01
   1.37394281e-02  -3.90780174e-03   2.43513397e-02   2.38629070e-02
   1.13014266e-02]

MSE Error Training
[[ 0.54992619]]

MSE Error Testing
[[ 0.65875293]]
```

Classification

Task 5

```
from sklearn import svm

def SVM_execute(X, y, C=1000):
    n = len(X)
    # Setting up training & Test data
    X_train = X[:n//2]
    X_test = X[n//2::]
    y_train = y[:n//2]
    y_test = y[n//2::]

    # Setting up the SVM
    clf = svm.SVC(C=C)
    clf.fit(X_train, y_train)
    # Retrieving predictions from trained SVM
    train_predictions = clf.predict(X_train)
    test_predictions = clf.predict(X_test)

    # Calculate accuracy
    percentage_train = sum([ y_train[i] == train_predictions[i] for i in range(n//2)
    percentage_test =  sum([ y_test[i] == test_predictions[i] for i in range(n//2)])

    print "\nAccuracy training"
    print percentage_train
    print "\nAccuray Test-set"
    print percentage_test

# Extracting wanted features
X = [[d['beer/ABV'], d['review/taste']] for d in data]
y = [d['beer/style'] == 'American IPA' for d in data]

SVM_execute(X, y)
```

Answer

```
Accuracy training
0.9226

Accuray Test-set
0.85632
```

Task 6

```
def feature_extractor(d):
    return [
        d['beer/ABV'],
        'IPA' in d['review/text']

    ]

X = [feature_extractor(d) for d in data]
y = [d['beer/style'] == 'American IPA' for d in data]

SVM_execute(X, y)
```

Answer

```
Accuracy training
0.94412

Accuray Test-set
0.95536
```

Using a feature vector cointaining the ABV and wether or not the review text contains ‘IPA’ in the review text gives the result

Accuracy training: 0.94412

Accuray Test-set: 0.95536

Task 7

The regularization constant penalizes the complexity model. With a high regularization constant penalizes high complexity models, forcing the SVM to choose a simpler model.

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

print "--\nRegularization constant = 0.1"
SVM_execute(X,y,0.1)

print "--\nRegularization constant = 10"
SVM_execute(X,y,10)

print "--\nRegularization constant = 1000"
SVM_execute(X,y,1000)

print "--\nRegularization constant = 100000"
SVM_execute(X,y,100000)
```

The accuracy from the different regularization constants:

Regularization Constant	**Training Data**	**Test data**
C=0.1	0.94508	0.95704
C=10	0.94416	0.95752
C=1000	0.94412	0.95536
C=100000	0.9442	0.9556

Task 8

The function f can be written as

$$\sum_{i=1}^{n} -log(1 + e^{-x_i \theta}) - X\theta - \lambda||\theta||_2^2$$

Where the second term is only considered if $y[i] = False$

```
from math import exp
from math import log
import random
import scipy
import numpy as np

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))
```

```
# 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]
  return -loglikelihood

# NEGATIVE Derivative of log-likelihood
def fprime(theta, X, y, lam):
  dl = [0.0]*len(theta)
  for i in range(len(X)):
    logit = - inner(X[i], theta)
    res = np.inner(X[i], exp(logit)) / ( 1 + exp(logit))
    if not y[i]:
        res -= X[i]
    dl += res
  for i in range(len(theta)):
    dl[i] -= 2*lam*theta[i]
  # Negate the return value since we're doing gradient *ascent*
  return np.array([-x for x in dl])


X = [[d['beer/ABV'], d['review/taste']] for d in data]
y = [d['beer/style'] == 'American IPA' for d in data]

X_train = X[:len(X)/2]
X_test = X[len(X)/2:]
y_train = y[:len(X)/2]
y_test = y[len(X)/2:]

# Run gradient descent
theta,l,info = scipy.optimize.fmin_l_bfgs_b(f, [0]*len(X[0]), fprime, args = (X_trai

predictions = [sigmoid(inner(X_test[i], theta)) > 0.5 for i in range(len(X_test))]
result = [ predictions[i] == y_test[i] for i in range(len(X_test))]
acc = sum(result) / float(len(result))

print "Final log likelihood =", f(theta,X,y,1.0)
print "Accuracy = ", acc
```

Answer

```
Final log likelihood = 14627.2048715
Accuracy =  0.9218
```

The final log-likelihood is 14627.2048715

The accuracy is 0.9218