Statistical Learning
Professor Keene

**Mini Project #3: Logistic Regression with Stochastic Gradient Descent**

Max Howald, Miraj Patel, Frank Longueira

This project uses the Wisconsin Breast cancer dataset found here:
https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Original%29
Specifically, we use the Wisconsin Diagnostic Breast Cancer dataset which comprises 569 samples. Each sample in the dataset is comprised of 30 real-valued features computed from an image of a fine needle aspirate (FNA) of a breast mass, and 1 label – the binary classification of the tumor in the image as either benign or malignant.

To begin, we divide the data into 300 training examples and 269 testing examples. Our goal is to classify the examples in the test set by training a logistic regression model using stochastic gradient descent (SGD). We normalize all data by scaling each feature column using its mean and standard deviation. This normalization ensures that the data in each column is of similar magnitude so no single feature would skew the model.

The project specified that we must implement the SGD algorithm ourselves. As a baseline, we first used the SGD classifier that is part of the scikit learn python package on our dataset. With default settings and an L2 penalty, the scikit learn classifier correctly classifies 95 – 97% of the examples in the test set.

Next we build our own SGD classifier. The classifier takes as parameters a learning rate η, and a stopping criteria (based on the change in total log likelihood between each iteration), and a maximum number of iterations. The classifier also supports either L1 or L2 regularization when calculating the log likelihood and gradient. The weights in the model are initialized to random values between 0 and 1.

After working out all the bugs, and experimenting with the learning rate and stopping criteria a bit, our classifier performs about the same as the scikit learn classifier, correctly classifying between 95 and 97% of the test set, depending on the random initialization and random shuffling of the training dataset.

Sample output and code follow:

Sample Output

```
$ ./classify.py wdbc.train wdbc.test

SKLEARN BUILT-IN CLASSIFIER RESULTS
Settings:  SGDClassifier(alpha=0.0001, average=False, class_weight=None,
epsilon=0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.15,
       learning_rate=’optimal’, loss=’log’;, n_iter=5, n_jobs=1,
       penalty=’l2’;, power_t=0.5, random_state=None, shuffle=True,
       verbose=0, warm_start=False)
Correctly classified malignant: 89
Incorrectly classified malignant: 2
Correctly classified benign: 170
Incorrectly classified benign: 8
TOTAL CORRECT: 259 / 269 = 0.96282527881


MY CLASSIFIER RESULTS
Trained for 300 iterations.
Learning rate: 0.005
Correctly classified malignant: 88
Incorrectly classified malignant: 3
Correctly classified benign: 173
Incorrectly classified benign: 5
```

```
#!/usr/bin/env python
import sys
import collections
import math
from sklearn.linear_model import SGDClassifier
from sklearn.preprocessing import StandardScaler
import numpy as np

NEGATIVE_CLASS = 'B' # Benign
POSITIVE_CLASS = 'M' # Malignant

# These parameters seemed to work well...
eta0 = 0.005
max_iterations = 300
stopping_val = 60
# Only set one (or 0) of these to true at a time...
L1_PENALTY = False
L2_PENALTY = True

def main():
    if len(sys.argv) != 3:
        print "Usage: " + sys.argv[0] + " <training file> <testing file>"
        exit(1)

    ### Parse and normalize the training data ###

    features, classes = parse_data(open(sys.argv[1], 'r'))
    scaler = StandardScaler()
    scaler.fit(features)
    scaled_features = scaler.transform(features)

    ### Train the baseline classifier ###
    clf = SGDClassifier(loss="log", penalty="l2")
    clf.fit(scaled_features, classes)

    ### Parse and normalize the test data ###
    test_features, test_classes = parse_data(open(sys.argv[2], 'r'))
    scaled_test_features = scaler.transform(test_features)

    ### classify and print stats for the baseline classifier ###
    print "SKLEARN BUILT-IN CLASSIFIER RESULTS"
    print "Settings: ", clf
    predictions = clf.predict(scaled_test_features)
    print_accuracy(test_classes, predictions)

    ### Train my classifier ###
    scaled_features_plus_intercept = np.ones((scaled_features.shape[0], scaled_f
eatures.shape[1] + 1))
    scaled_features_plus_intercept[:, 1:] = scaled_features
    weights, t = train(scaled_features_plus_intercept, classes)

    ### Test my classifier ###
    print "\n\nMY CLASSIFIER RESULTS"
    print "Trained for", t, "iterations."
    print "Learning rate:", eta0
    scaled_test_features_plus_intercept = np.ones((scaled_test_features.shape[0]
, scaled_test_features.shape[1] + 1))
    scaled_test_features_plus_intercept[:, 1:] = scaled_test_features
    my_predictions = [classify(weights, instance) for instance in scaled_test_fe
atures_plus_intercept]
    print_accuracy(test_classes, my_predictions)

def train(X, classes):
    # randomly initialize weights between 0 and 1.
    weights = np.random.rand((X.shape[1]), 1)
    diff = stopping_val + 1.

    total_ll = total_log_likelihood(X, classes, weights)

    t = 0
    rows = range(len(X))
    while (diff > stopping_val) and (t < max_iterations):
        for i in xrange(len(X)):
            log_prob = 1 / (1 + np.exp(-X[i,:].dot(weights)))
            error = classes[i] - log_prob
            error_product = (X[i, :] * error).reshape(X.shape[1],1)
            weights = weights + eta0 * error_product

        new_ll = total_log_likelihood(X, classes, weights)
        diff = np.abs(new_ll - total_ll)
        total_ll = new_ll

        np.random.shuffle(rows)
        X = X[rows, :]
        new_classes = [None] * len(classes)
        for i in xrange(len(classes)):
            new_classes[i] = classes[rows[i]]
        classes = new_classes
        t += 1
    return weights, t


def total_log_likelihood(X, Y, W):

    probs = 1 / (1 + np.exp(-X.dot(W)))
    ones_arr = np.ones((1, probs.shape[1]))

    # epsilon avoid log of negatives or 0...
    epsilon = 1e-24
    log_likelihoods = Y * np.log(probs + epsilon) + (ones_arr - Y) * np.log(ones
_arr - probs + epsilon)
    total_ll = -1 * log_likelihoods.sum()

    if L1_PENALTY:
        total_ll += np.abs(W).sum()
    if L2_PENALTY:
        total_ll += np.power(W, 2).sum() / 2
    return total_ll


# calculate the probability and classify as malignant if p > 0.5
def classify(W, X):
    return 1 if 1 / (1 + np.exp(-X.dot(W))) > 0.5 else 0

def print_accuracy(ground_truth, predictions):
    true_pos, true_neg, false_pos, false_neg = 0,0,0,0
    for t, p in zip(ground_truth, predictions):
        true_pos += (t == 1 and p == 1)
        false_pos += (t == 1 and p == 0)
        true_neg += (t == 0 and p == 0)
        false_neg += (t == 0 and p == 1)

    print "Correctly classified malignant:", true_pos
    print "Incorrectly classified malignant:", false_pos
    print "Correctly classified benign:", true_neg
    print "Incorrectly classified benign:", false_neg
    print "TOTAL CORRECT:",  ( true_pos + true_neg ), "/", len(predictions), "="
, float(true_pos + true_neg) / float(len(predictions))


def parse_data(input_stream):
    classes = []
    features = []
    for line in input_stream:
        fields = line.strip('\n').split(',')
        if fields[1] == NEGATIVE_CLASS:
            classes.append(0)
        else:
            classes.append(1)
        features.append(map(float, fields[2:]))

    return features, classes


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