Katherine Olson

STA 141C Homework #3

This assignment works on classification and regression using datasets listed in https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/ . This data is in "SVMLight" format. Unfortunately, we were given no information about the datasets and what exactly we are predicting.

Problem 1

1. Ridge regression

Problem:

This problem uses the cpusmall_scale dataset. The dataset is randomly split into testing and training with 80% training and 20% testing. The goal of this problem is to solve the following equation for ridge regression:

$$w^* = f(w) = \underset{w}{\operatorname{argmin}}\{\tfrac{1}{2}\textstyle\sum_{i=1}^{n}(x_i^T w - y_i)^2 + \tfrac{\lambda}{2}\|w\|^2\},$$

where $x_i \in \mathbf{R}^d$ is the ith training sample, and $y_i \in \mathbf{R}$ is the ith target value. Set $\lambda = 1$.

Then find the mean squared error of the solution:

$$\frac{1}{n_{test}}\sum_{i=1}^{n_{test}}(x_i^T w^* - y_i)^2$$

Code snippets:

My main function reads in the data, parses it, splits it into testing and training, and sends it to the ridge regression function for each level of lambda.

The ridge regression function seen below gets X and y from the training set of data and uses it to calculate w*. w* is calculated through the closed form solution to the ridge regression equation that uses the normal equations. Then function then gets X and y from the testing data and finds the mean squared error when using w* on that data.

```
def RidgeRegression(train, test, lamda):
    """
    Input: Training data, testing data, and a value for lambda
    Output: The mean squared error of the ridge regression
    """
    # Get x and y from the training data:
    y = np.array(train[0])
    x = train.drop(0, 1).as_matrix()

    # Get beta from the normal equations / closed form solution:
    wStar = np.linalg.inv(x.T.dot(x) + lamda * np.eye(12)).dot(x.T.dot(y))
```

```
        # Get x and y from the testing data:
        y = np.array(test[0])
        x = test.drop(0, 1).as_matrix()

        # Get an error estimate of using that beta on the testing data:
        MSE = (1.0 / test.shape[0]) * sum((x.dot(wStar) - y) ** 2)
        return MSE
```

2. Results:

λ	MSE
0.01	101.510
0.1	101.540
1	101.832
10	104.964
100	126.620

So, the lowest MSE comes from $\lambda = 0.01$.
Then, the MSE slowly increases and jumps at 100.

3. Gradient Descent

Problem: Write the gradient descent algorithm with fixed step size. Run it on the "E2006-tfidf" data with $\in = 0.001$.

Code snippets:

My main function reads in the two E2006 data files using the load_svmlight_file function from sklearn.datasets using the the parameter n_features = 150360 to make sure all lines are read in. The function then loops through each step size and prints out the mean squared error for running gradient descent on the E2006 data and the CPU data with the following function:

```
def GradDesc(x, y, xtest, ytest, stepSize, stopCond):
        """
        Input: The x and y matrices / arrays for both testing and training.
                The step size and the stopping condition.
        Output: The mean squared error of the beta estimated by the gradient descent.
        """
        # Start with a random vector and convert y to sparse:
        w = sparse.csr_matrix(np.random.rand(x.shape[1])).T
        y = sparse.csr_matrix(y).T
        # Get norm of derivative:
        r0 = scipy.sparse.linalg.norm(x.T.dot(x.dot(w) - y) + w)

        iterations = 0
        while iterations < 50: # Run for at most 50 iterations
                # Get the gradient:
                g = x.T.dot(x.dot(w) - y) + w

                # If it has converged, then stop:
                if scipy.sparse.linalg.norm(g) <= stopCond * r0:
                        break

                # Compute the new w with the step size and continue on:
                w = w - (stepSize * g)
                iterations += 1

    # Compute the mean squared error for the final w:
        ytest = sparse.csr_matrix(ytest).T
        res = xtest.dot(w) - ytest
        MSE = (1.0 / xtest.shape[0]) * sum(res.multiply(res))

        return MSE.data[0] # Grad value from inside matrix
```

4. Results - using $\lambda = 1$ and $\in = 0.001$:

The E2006 Data	
Step Size	**MSE**
10^{-2}	inf
10^{-3}	5.395e+228
10^{-4}	7.896e+125
10^{-5}	0.165
10^{-6}	0.166
10^{-7}	0.172

The gradient converges with a step size smaller than 10^{-4}. For those values, the mean squared error is close to zero and much smaller than the results of ridge regression. The best results are from the step size 10^{-5}.

<u>Problem 2:</u>

This problem is to write the code for logistic regression. The training data consists of feature vectors x_i and labels (1 or -1) y_i. The logistic regression model is obtained from solving

$$w^* = f(w) = \underset{w}{\operatorname{argmin}}\{\sum_{i=1}^{n} \log(1 + e^{-y_i w^T x_i}) + \frac{\lambda}{2}\|w\|^2\}$$

1. Derive the gradient for logistic regression:

<u>Problem:</u> Show how the gradient for logistic regression is defined.

<u>Solution:</u>

Starting with the formula:

$$f(w) = \underset{w}{\operatorname{argmin}}\{\frac{1}{n}\sum_{i=1}^{n} \log(1 + e^{-y_i w^T x_i}) + \frac{\lambda}{2}\|w\|^2\}$$

Looking at the individual part of the sum for each i gives:

$$f_i(w) = \log(1 + e^{-y_i w^T x_i}) + \frac{\lambda}{2}\|w\|^2$$

Then, to get the gradient, we need the derivative of $f_i(w)$ with respect to each w_i:

$$\frac{\partial f_i(w)}{\partial w_1} = \frac{-y_i x_{i1} e^{-y_i w^T x_i}}{1 + e^{-y_i w^T x_i}} + \lambda w_1 = x_{i1}(\frac{-y_i e^{-y_i w^T x_i}}{1 + e^{-y_i w^T x_i}}) + \lambda w_1$$

$$\frac{\partial f_i(w)}{\partial w_2} = \frac{-y_i x_{i2} e^{-y_i w^T x_i}}{1 + e^{-y_i w^T x_i}} + \lambda w_2 = x_{i2}(\frac{-y_i e^{-y_i w^T x_i}}{1 + e^{-y_i w^T x_i}}) + \lambda w_2$$

$$\cdots$$

$$\frac{\partial f_i(w)}{\partial w_n} = \frac{-y_i x_{in} e^{-y_i w^T x_i}}{1 + e^{-y_i w^T x_i}} + \lambda w_n = x_{in}(\frac{-y_i e^{-y_i w^T x_i}}{1 + e^{-y_i w^T x_i}}) + \lambda w_n$$

Thinking about this set of derivatives in terms of matrix multiplication gives one equation:

$$f(w) = x_i(\frac{-y_i e^{-y_i w^T x_i}}{1 + e^{-y_i w^T x_i}}) + \lambda w$$

This is the gradient!

2. Run gradient descent on the data with that data for a fixed step size

<u>Problem:</u> Write the code for gradient descent with logistic regression and a fixed step size.

<u>Code snippets:</u>

My main function reads in the data, splits it into testing and training, and then loops through each step size and calls the following function:

```
def LogRegFixed(x, y, xtest, ytest, stepSize, stopCond):
    """
    Input: Training x and y, testing x and y, a step size, and a stopping condition
    Output: The mean squared error of the logistic regression gradient descent prediction on
    the testing data
    """
    # Start with a random vector:
    w = sparse.csr_matrix(np.random.rand(x.shape[1])).T
    y = sparse.csr_matrix(y)

    # Get norm of derivative:
    r0 = scipy.linalg.norm(LogRegComputeGrad(y, x, w, 1))

    iterations = 0
    while iterations < 50: # Run for at most 50 iterations
        # Get the gradient:
        g = LogRegComputeGrad(y, x, w, 1)

        # If it has converged, then stop:
        if scipy.linalg.norm(g) <= stopCond * r0:
            break

        # Compute the new w with the step size and continue on:
        w = sparse.csr_matrix(w - (stepSize * g))
        iterations += 1

    # Compute the mean squared error for the final w:
    ytest = sparse.csr_matrix(ytest)

    pred = np.sign(w.T.dot(xtest.T).todense())
    same = pred == ytest
    return float(same.sum()) / ytest.shape[1]
```

To get the gradient for each w, the above function calls the function below:

```
def LogRegComputeGrad(y, x, w, lamda):
    """
    Input: The y, x, and w matrices. The value of lambda.
    Output: The gradient for those matrices for logistic regression.
    """
    # The part of the equation that is the e^-yiwTxi:
    expPart = np.exp(y.multiply(w.T.dot(x.T)).todense() * -1.0)

    # The part of the equation in the fraction:
    frac = y.multiply(expPart * -1) / (expPart + np.ones(expPart.shape[1]).T )
```

```
        # Adding it all together:
        total = x.T.dot(frac.T) + (w * lamda)

        return total
```

Results for Gradient Descent Fixed on News20 Logistic Regression	
Step Size	**Accuracy**
10^{-2}	0.831
10^{-3}	0.515
10^{-4}	0.493
10^{-5}	0.493
10^{-6}	0.493
10^{-7}	0.493

I get the highest accuracy with a step size of 10^{-2}, then the accuracy decreases significantly for smaller step sizes. This highest accuracy is only 83%, which is okay, but not great.

3. Gradient Descent with Line Search

Problem: Write the code for gradient descent with line search for logistic regression.

Code snippets:

The main function from part 2 sends the data to the function that performs gradient descent with line search:

```
def LogRegLine(x, y, xtest, ytest, lamda, stopCond):
        """
        Input: Training x and y, testing x and y, lambda, and the stopping condition
        Output: The accuracy for gradient descent with line search using logistic regression
        """
        # Start with a random vector:
        w = sparse.csr_matrix(np.random.rand(x.shape[1])).T
        y = sparse.csr_matrix(y)

        # Get norm of derivative:
        r0 = scipy.linalg.norm(LogRegComputeGrad(y, x, w, lamda))

        iterations = 0
```

```
    while iterations < 50: # Run for at most 50 iterations
        # Get the gradient:
        g = LogRegComputeGrad(y, x, w, 1)

        # If it has converged, then stop:
        if scipy.linalg.norm(g) <= stopCond * r0:
            break

        stepSize = 1.0

        # Find a good step size:
        while f(x, y, sparse.csr_matrix(w - (stepSize * g)), lamda) >= f(x, y, w, lamda):
            stepSize = stepSize / 2.0

        # Update w and  increase the number of iterations:
        w = sparse.csr_matrix(w - (stepSize * g))
        iterations += 1

    # Compute the mean squared error for the final w:
    ytest = sparse.csr_matrix(ytest)

    pred = np.sign(w.T.dot(xtest.T).todense())
    same = pred == ytest
    return float(same.sum()) / ytest.shape[1]
```

This function calls the following function to compute f(w):

```
def f(x, y, w, lamda):
    """
    Input: The training x and y, the current w, and lambda
    Output: The value for f(w) where f(w) = the logistic regression function
    """
    expPart = np.exp(y.multiply(w.T.dot(x.T)).todense() * -1.0)
    inBrakets = np.log(1 + expPart).sum() + ((lamda / 2.0) * (scipy.sparse.linalg.norm(w)
                                                                                   **2))
```

Results for Gradient Descent Line Search on News20 Logistic Regression	
λ	**Accuracy**
10^{-1}	0.95850
10^{-2}	0.95925
10^{-3}	0.95750
10^{-4}	0.95975
10^{-5}	0.95800
10^{-6}	0.96025

This method has very accurate results. The smallest lambda gives the best results, but the accuracy barely differs for each lambda.

Problem 3:

Compare the results to functions in scikit-learn.

Ridge Regression

I have the testing ridge regression function start the same way as the main function in problem 1, and then I save the training data into x and y and the testing data into xt and yt. I call the Ridge function from scikit-learn with this loop:

```
for lamda in [0.01, 0.1, 1.0, 10.0, 100.0]:
        model = Ridge(alpha = lamda)
        model.fit(x, y) # Fit with the training data
        wStar = model.predict(xt) # Get the predictions for testing
        mse = (1.0 / test.shape[0]) * sum((wStar - yt) ** 2)
        ridgeResults = ridgeResults.append(pd.DataFrame.from_dict([{"Lambda" : lamda, "MSE
                                                                         Scikit-learn" : mse}]))
```

λ	**MSE: My Results**	**MSE: Scikit-learn**
0.01	101.510	98.591
0.1	101.540	98.611
1	101.832	98.843
10	104.964	102.143

100	126.620	128.452

The scikit-learn function does slightly better than my function for the first four values of $\boldsymbol{\lambda}$. Then for $\boldsymbol{\lambda} = 100$ my function has a lower MSE. The results for each $\boldsymbol{\lambda}$ differ by about 2 or 3 points of MSE.