HW3   Daniel Rozen drozen3

1. Write down detailed formulas for the gradient of the loss function in the case of logistic regression

DAVA HW 3

1. Logistic regression model

$$p_\theta(Y=y^{(i)}|X=x^{(i)}) = \frac{1}{1+e^{(y^{(i)}\langle\theta,x^{(i)}\rangle)}}$$ → definition of the logistic regression model

$$\hat{\theta}_{MLE} = \arg\max_\theta \prod_{i=1}^n p_\theta(Y=y^{(i)}|X=x^{(i)})$$

$$= \arg\max_\theta \log\left(\prod_{i=1}^n p_\theta(Y=y^{(i)}|X=x^{(i)})\right)$$

$$= \arg\max_\theta \sum_{i=1}^n \log\left(p_\theta(Y=y^{(i)}|X=x^{(i)})\right)$$

Substitute in $p(y|x)$

$$\hat{\theta}_{MLE} = \arg\max_\theta \sum_{i=1}^n \log\left(\frac{1}{1+e^{(y^{(i)}\langle\theta,x^{(i)}\rangle)}}\right)$$ Since $\log(\frac{1}{a}) = -\log(a)$

$$= \arg\max_\theta \sum_{i=1}^n -\log(1+e^{(y^{(i)}\langle\theta,x^{(i)}\rangle)})$$

$$= \arg\min_\theta \sum_{i=1}^n \log(1+e^{(y^{(i)}\langle\theta,x^{(i)}\rangle)})$$

Compute the gradient:

$$\hat{\theta}_{MLE} = \arg\min_\theta L(\theta) \quad \text{where } L(\theta) = \sum_{i=1}^n \log(1+\exp(y^{(i)}\langle\theta,x^{(i)}\rangle))$$

In order to compute $\nabla L(\theta)$ we need to calculate $\frac{\partial}{\partial\theta^{(j)}} L\theta$

j ranges from 1 to d, where d = # of features
i " " " to n, where n = # of cases in the dataset

$$\frac{\partial}{\partial\theta^{(j)}} L(\theta) = \frac{\partial}{\partial\theta^{(j)}} \sum_{i=1}^{n} \log(1+\exp(y^{(i)}\langle\theta, x^{(i)}\rangle))$$

$$= \sum_{i=1}^{n} \frac{\partial}{\partial\theta^{(j)}} \log(1+\exp(y^{(i)}\langle\theta, x^{(i)}\rangle))$$

$\left(\text{Since } \frac{d}{dx}\ln(f(x)) = \frac{f'(x)}{f(x)}\right)$

$$= \sum_{i=1}^{n} \left(\frac{\frac{\partial}{\partial\theta^{(j)}}(1+\exp(y^{(i)}\langle\theta, x^{(i)}\rangle)}{1+\exp(y^{(i)}\langle\theta, x^{(i)}\rangle)} \right)$$

$$= \sum_{i=1}^{n} \left(\frac{\frac{\partial}{\partial\theta^{(j)}}(\exp(y^{(i)}\langle\theta, x^{(i)}\rangle)}{1+\exp(y^{(i)}\langle\theta, x^{(i)}\rangle)} \right)$$

Since $\exp(y^{(i)}\langle\theta, x^{(i)}\rangle) = \exp\left(y^{(i)}\sum_{k=1}^{d}\theta_k x_k^{(i)}\right)$

and $\frac{d}{dx}\exp(f(x)) = \exp(f(x)) \cdot f'(x)$ $\therefore$

$$\frac{\partial}{\partial\theta^{(j)}} L(\theta) = \sum_{i=1}^{n} \frac{\exp(y^{(i)}\langle\theta, x^{(i)}\rangle) \cdot \frac{\partial}{\partial\theta^{(j)}}\left(y^{(i)}\sum_{k=1}^{d}\theta_k x_k^{(i)}\right)}{1+\exp(y^{(i)}\langle\theta, x^{(i)}\rangle)}$$

Since $\frac{\partial}{\partial\theta^{(j)}}$ is w.r.t. j when $k \neq j$ $\theta_k x_k^{(i)}$ is a constant & $\therefore$ its derivative will be 0

$\therefore$ we only need to compute the derivative for $k = j$

$$\therefore \frac{\partial L(\theta)}{\partial \theta_j} = \sum_{i=1}^{n} \frac{\exp(y^{(i)}\langle\theta, x^{(i)}\rangle)\cdot \frac{\partial}{\partial\theta^{(i)}}(-y^{(i)}\theta_j x_j^{i})}{1+\exp(y^{(i)}\langle\theta,x^{(i)}\rangle)}$$

$$= \sum_{i=1}^{n} \frac{\exp(y^{(i)}\langle\theta,x^{(i)}\rangle)\, y^{(i)} x_j^{(i)}}{1+\exp(y^{(i)}\langle\theta,x^{(i)}\rangle)} = \sum_{i=1}^{n} \frac{y^{(i)}x_j^{(i)}}{1+\exp(-y^{(i)}\langle\theta,x^{(i)}\rangle)}$$

$\nabla L(\theta) = \left(\frac{\partial L(\theta)}{\partial\theta_1}, \dots, \frac{\partial L(\theta)}{\partial\theta_d}\right)$ where j ranges from 1 to d.

and write detailed pseudo code for training a LR model based on gradient descent.

(a) initialize the dimensions of theta to random values

```
for j = 1… d
        $\theta_j$ = random value
```

(b) for j = 1… d (for every dimension of vector theta)
update

Gradient descent update rule

$$\theta_j \leftarrow \theta_j - \alpha\left(\sum_{i=1}^{n} \frac{y^{(i)}x_j^{(i)}}{1+\exp(y^{(i)}\langle\theta,x^{(i)}\rangle)}\right)$$

```
for j = 1… d:
        d: sum = 0
        for i = 1 … n:
                a= $\exp(-y^i < \theta, x^i >)$
```

$$\text{sum} = \text{sum} + \frac{y^i x_j^i}{1+a}$$

θ_j = θ_j - α * sum

(c) repeat step (b) until the updates become smaller than a threshold: delta or time.

d) for every m iterations, alpha = sqrt(alpha)

Count how many operations are done per each gradient descent iteration and explain how you computed your answer (use the following variables in your answer: n for the number of examples and d for the dimensionality).

Step a takes d steps as it loops through d steps to give θ_j a random value. But this is only done in the 1st iteration.

Step b take d * n steps as it loops through all j = 1… d and updates θ_j with a sigma that loops through i = 1 … n:

Each step of the *a* variable is 1 exponent and 1 multiplication and 1 inner product = d multiplications and d additions. For a total of 1 exponent and d+1 multiplications and d additions.

Each step of sum is 1 addition + 1 divide + 1 multiplication

Each final update of θ_j is one multiplication and 1 subtraction.

For a total of each gradient descent:

d*n*(1 exponents, d+2 multiplications + d+1 additions + 1 divide)
+ d * (1 mult + 1 subtraction)

= dn exponents + (ddn +3dn +d) multiplications + (ddn + dn +d) additions/subtractions + dn divides

2. Implement in R logistic regression based on gradient descent. To avoid unnecessary slowdown use vectorized code when computing the gradient (avoid loops).

See code

*3.	Train and evaluate your code on the BreastCancer data from the mlbench R package. Specifically, randomly divide the dataset into 70% for training and 30% for testing and train on the training set and report your accuracy (fraction of times the* *model made a mistake) on the train set and on the test set. Repeat the random partition of 70% and 30% 10 times and average the test accuracy results over the 10 repetitions.*

Using the parameters
theta = logReg(trainX, trainY, alpha=.01, decay = 0.99, threshold=0.0001, maxIter=30000)

achieved the following results (in %)

```
> AverageTrainError
[1] 2.92887
> AverageTestError
[1] 3.414634
```

Try several different selections of starting positions - did this change the parameter value that the model learned?

Changing the starting random theta only slightly changed the parameter values that the model learned, as we see from the following results over 10 repetitions, done twice:

```
> AverageTrainError
[1] 2.845188
> AverageTestError
[1] 3.804878
```

```
> AverageTrainError
[1] 2.656904
> AverageTestError
[1] 3.170732
```

The theta percentage difference was

```
0.05439207%
```

Try to play with different convergence criteria to get better accuracy.

With the following parameters

theta = logReg(trainX, trainY, theta, alpha=.001, decay = 0.99, threshold=0.000001, maxIter=30000)

```
> AverageTrainError
[1] 3.221757
> AverageTestError
[1] 3.365854
```

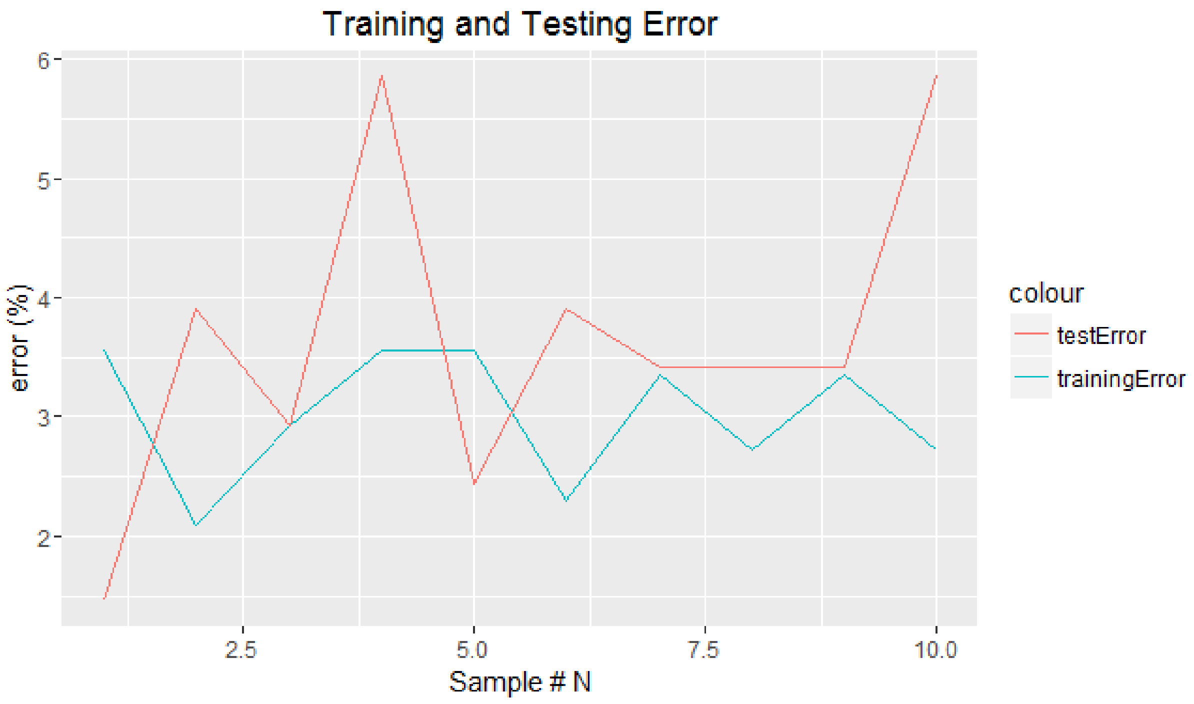
theta = logReg(trainX, trainY, theta, alpha=.001, decay = 0.99, threshold=1e-8, maxIter=30000)

```
> AverageTrainError
```

```
[1] 3.012552
> AverageTestError
[1] 3.219512
```

theta = logReg(trainX, trainY, theta, alpha=.01, decay = 0.99, threshold=1e-8, maxIter=30000)

```
> AverageTrainError
[1] 3.012552
> AverageTestError
[1] 3.658537
```

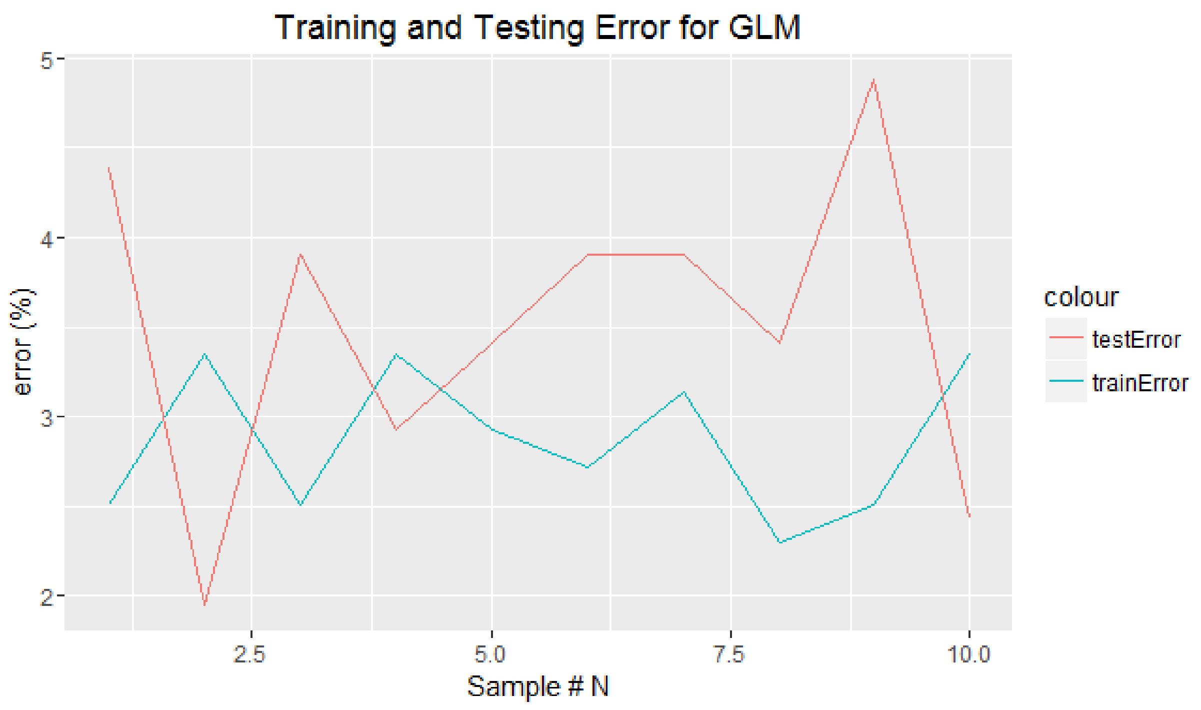


Changing the convergence criteria (alpha's and the threshold's) didn't appear to have much effect on the error.

This is likely because the gradient descent algorithm will reach the single global maximum regardless of the starting point.

4. Repeat (3) but this time using logistic regression training code from an R package such as glm2.

How did the accuracy in (4) compare to the accuracy in (3).



The accuracies are very similar as glm2 averaged over 10 repetitions gives us:

```
> GLMAverageTrainError
[1] 2.866109
 > GLMAverageTestError
[1] 3.512195

As compared to
```

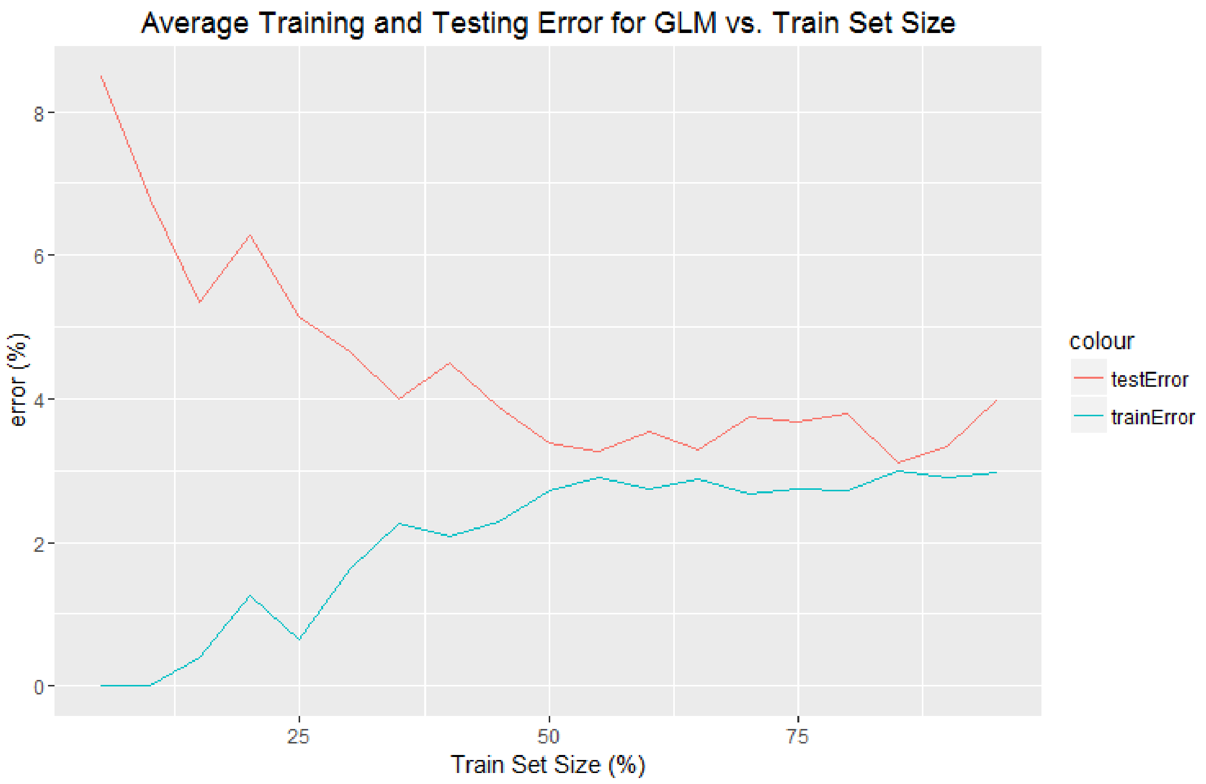
theta = logReg(trainX, trainY, theta, alpha=.001, decay = 0.99, threshold=1e-8, maxIter=30000)

```
> AverageTrainError
[1] 3.012552
> AverageTestError
[1] 3.658537
```

5 Repeat (4), but replace the 70%-30% train-test split with each of the following splits: 5%-95%, 10%-90%, …, 95%-5%. Graph the accuracy over the training set and over the testing set as a function of the size of the train set. Remember to average the accuracy over 10 random divisions of the data into train and test sets of the above sizes so the graphs will be less noisy.

```
> glmAvResultsDF
   trainSetSize trainError testError
1             5  0.0000000  8.489985
2            10  0.0000000  6.780488
3            15  0.3921569  5.352840
```

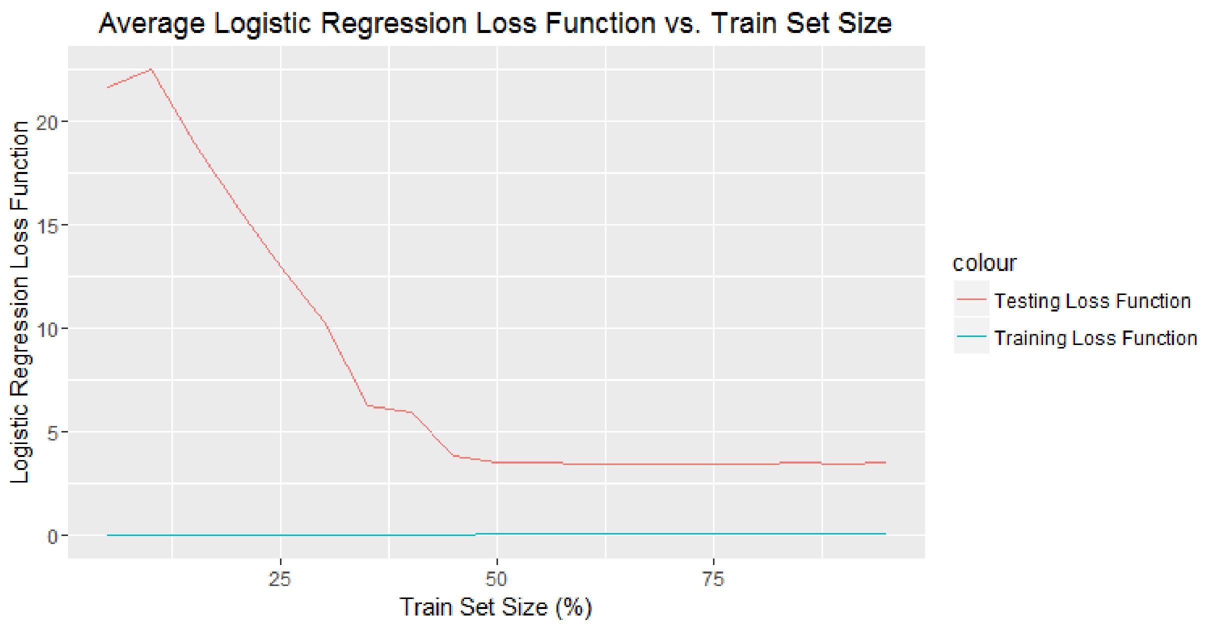
```
4           20  1.2500000  6.288848
5           25  0.6470588  5.146199
6           30  1.6176471  4.655532
7           35  2.2594142  4.009009
8           40  2.0879121  4.512195
9           45  2.2801303  3.882979
10          50  2.7272727  3.391813
11          55  2.9066667  3.279221
12          60  2.7383863  3.540146
13          65  2.8893905  3.291667
14          70  2.6778243  3.756098
15          75  2.7343750  3.684211
16          80  2.7106227  3.795620
17          85  3.0000000  3.106796
18          90  2.8990228  3.333333
19          95  2.9629630  4.000000
```



From the above graph, we see that test Error starts off quite high at around 8.5%, and train error extremely low at 0%. This is likely due to overfitting, as little training data is available (only 34 points). Therefore the training data will be fit very well, but not the test data. The test error decreases and reaches a minimum, while simultaneously, the train error increase and reaches a maximum at around 50% train set. This is likely the point where overfitting is no longer an issue.

6. Repeat (5) but instead of graphing the train and test accuracy, graph the logistic regression loss function (negative log likelihood) over the train set and over the test set as a function of the train set size.

```
> glmAvResultsDF
   trainSetSize lossFunTrain lossFunTest
1             5 4.953533e-12   21.694392
2            10 2.563960e-12   22.520084
3            15 1.870302e-02   18.965054
4            20 2.619238e-02   15.790189
5            25 3.667772e-02   13.001083
6            30 4.310808e-02   10.369629
7            35 6.083520e-02    6.294566
8            40 6.558593e-02    5.945279
9            45 7.193388e-02    3.847661
10           50 7.589560e-02    3.577615
11           55 7.562096e-02    3.516306
12           60 7.679434e-02    3.466438
13           65 7.368755e-02    3.466434
14           70 7.547269e-02    3.472905
15           75 7.575261e-02    3.442769
16           80 7.684441e-02    3.461733
17           85 7.835788e-02    3.515948
18           90 7.675557e-02    3.502877
19           95 7.525636e-02    3.528500
>
```



We see that Testing Loss Function converges to a minimum at around Train Set Size = 50%. This makes sense as the testing loss should start of high with a small sample to train on, and then decrease as the training sample increases relative to the test sample.   Once Train Set Size is close to Test Set Size, the issues of overfitting are overcome.

Also The training loss should be consistently low as it's testing on the same sample it trained on which is what we see in the graph.