CSE 474: Project 1
Logistic Regression

Peter VanNostrand

09/25/2019

EXPERIMENTAL SETUP

To perform logistic regression on the cancer dataset, I created the file `project1.py` which reads the dataset file, trains a logistic classifier and then tests that classifier on a subset of the dataset. Once complete accuracy metrics are printed to the console and a plot of the training loss is displayed.

Functional Description:

When executed the first thing this program does is load the Wisconsin Diagnostic Breast Cancer dataset from the `wdbc.dataset` file. This is accomplished using the load_dataset() function which takes in a path to the dataset file and returns two numpy arrays X and Y which contain the features for each sample and the labels for each sample respectively. The features of X are normalized to the range [0,1] to avoid overflows in later calculations. Next the partition_data() function is used to randomly shuffle the matrices X and Y (preserving correspondence between the two) and produce two pairs of features and labels to be used for training and testing. Once the data is prepared the train() function is used to train a logistic regression classifier using gradient descent on the training samples. This produces a set of weights for each feature and a bias term. During training the cost is stored at each epoch and plotted. Finally, the trained weights and bias are used to perform inference on the testing dataset and the accuracy, precision, and recall of the model are calculated.

Notes:

1. The dataset file should be named `wdbc.dataset` and be located in the same directory as `project1.py`
2. The following python libraries should be installed
    a. numpy
    b. matplotlib
    c. sys
    d. csv

HYPER-PARAMETER TUNING

During the training of this logistic classifier the two most important hyper-parameters are number of epochs and learning rate. In order to determine the optimal value of these parameters I created the hyper_tune() function. This function performs training and validation for a wide range of epochs and learning rates and stores the results of each to a csv file. This process is very computationally intensive as the classifier is trained many times. As such this process has a significant runtime and is not included in the normal execution of this program.

After running the hyper_tune() function I examined the generated csv file and picked a handful of epoch and learning rate pairs that were representative of different regimes. First we can examine the effect of learning rate on accuracy, precision, and recall as shown in the table below.

Learning Rate

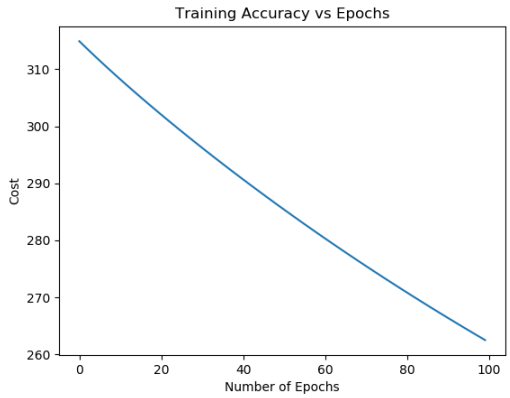
Table 1: Effect of Learning Rate on Performance

Epochs	LR	Accuracy	Recall	Precision	Sum
99900	0.002	0.955752	0.947368	0.923077	2.826198
99900	0.05	0.99115	1	0.974359	2.965509
99900	0.1	0.982301	1	0.973684	2.955985
99900	0.15	0.99115	0.973684	1	2.964835
99900	0.25	0.982301	0.973684	0.973684	2.929669

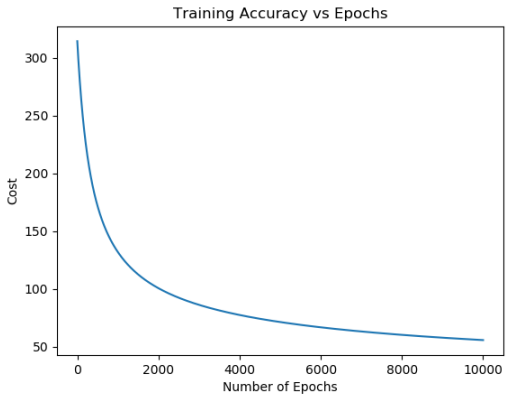
For this test I intentionally used a very high number of epochs to ensure that differences in metrics were due to the changing learning rate. To accurately measure the overall performance of the classifier the last column represents the arithmetic sum of the accuracy, precision, and recall. As we can see in the table the classifier has poorer performance for lower learning rates, then increases in performance with growing learning rate up to a point where performance begins to drop again. This is because at high learning rates the gradient descent becomes too steep meaning that changes in the weights are not able to be fine-tuned well. However, it is worth noting that all learning rates performed relatively well in this test due to the very high number of iterations, decreasing the number of iterations did not significantly affect the relative performance of these learning rates. **Based upon this data I chose a learning rate of 0.05**

Number of Epochs

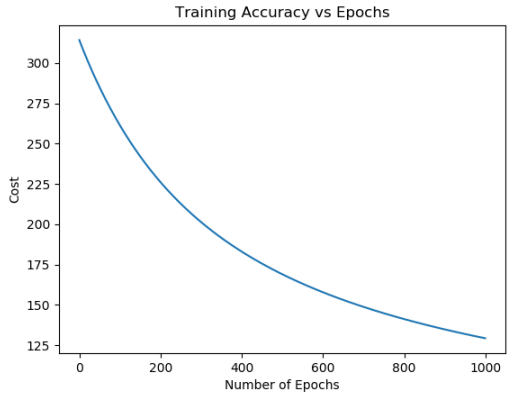
Having selected a learning rate, I ran the logistic classifier for several different epoch values starting with 100 and increasing to 100,000 and examined the graph of cost vs epoch for each. These plots are shown below



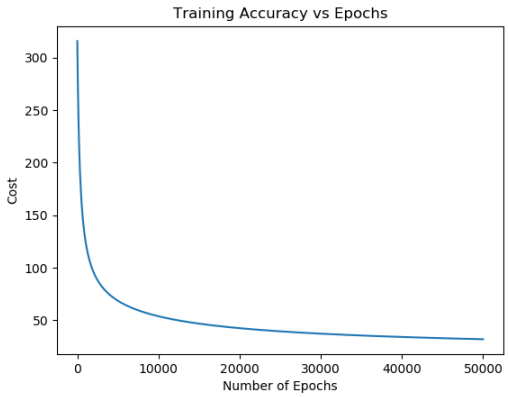
At 100 epochs we can see that the model is becoming increasingly accurate with each epoch in a somewhat linear fashion.



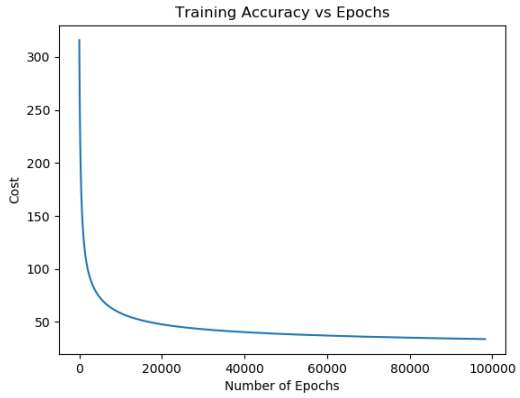
At 10,000 epochs we can see that the accuracy has levelled off further, meaning that the logistic regression classifier has learned a set of weights that produces relatively accurate predictions for the training data



At 1,000 epochs we can see that the accuracy is beginning to level off in an exponential fashion. As per the law of diminishing returns additional epochs are yielding smaller and smaller increases in accuracy



By 50,000 epochs the accuracy has almost entirely stopped increasing and training is beginning to take several seconds as the number of operations continues to increase



Finally, at 100,000 epochs the accuracy is practically the same as at 50,000 epochs and the runtime has become somewhat annoyingly slow.

Based on the data above, I selected **50,000 as the optimal number of epochs** as this ensures that the classifier learns a very good set of weights without taking an impractically long time to produce only marginally better results.

RESULTS

Using the optimal values found above ($learning\_rate = 0.05$ and $epochs = 50{,}000$) I performed training and validation on the WDBC dataset and found the following performance metrics

Accuracy (%)	Precision (%)	Recall (%)
99.12	100.00	97.50

These values vary slightly depending on the randomized distribution of samples between the training and testing subsets, but are regularly in the high 90's