CSE 250B Project 1

Lund, Sigurd Stoeen
sigurlu@stud.ntnu.no

Gilani, Ryden
mistafrowns@gmail.com

Wolff, Thomas
thomawo@stud.ntnu.no

January 23, 2014

Introduction

In a team of three students, we conducted a project whose goal was "to understand logistic regression and gradient-based optimization and practical issues that arise in classifier training.". In order to achieve this goal we were asked to implement Stochastic Gradient Descent (SGD) for learning logistic regression models with L_2 regularization, in addition to using an already implemented version of Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS)[1]. We were given a training set to use for learning, and a set of test data for evaulating our result. Each example in the data sets consists of a 800 dimensional feature vector, and a binary classification. The objective of the SGD algorithm is to learn the mapping from the examples feature vector to its classification. This mapping is assumed to be some linear combination of the values of the features, and every example in the data sets are assumed to satisfy it. To be able to assess how well our learned model estimates the actual model, every example in the test set is classified using our learned model, and compared to the actual classification of the example.In order to optimize our algorithms performance and accuracy we performed a grid search over its hyperparameter space. Our implementation yielded a model with an error rate of 10%.

Design and Analysis of Algorithms

The SGD is given as input a data set of N observations, where each observation is a vector of D dimensions and a label L. Its goal is to estimate the actual function mapping the vector to the label. SGD works by iterating over the different training examples, and calculating a weight β_j of each feature x_j. After going through all training examples, the model learned is evaluated by predicting new data and calculate log conditional likelihood (LCL). This process is repeated until the LCL converges.

Randomizing data

The data from the training set is shuffled randomly after being read, making sure that the ordering is random. This is done by the shuffle method in the random library of Python[2]

Training using SGD

Each iteration of training takes one random sample, and calculates the probability for that example to be labeled with 1. This probability is calculated by summing all features for that example multiplied with the weight of the feature:

$$z = \sum_{j=1} x_j \beta_j$$

Since the probability should be between 0 and 1, a sigmoid function is used:

$$p = \frac{1}{1 + \exp(-z)}$$

Then all β-values are updated based on the following calculation using L_2 regularization:

$$\beta_j = \beta_j + \lambda((y - p)x_j - 2\mu * \beta_j)^{[3]}$$

As the equation describes, the update of the β-values, are depending on the difference between the true label y, and the probability for the label to be 1, p. This is in fact the reason that the algorithm converges to a optimal model, since the β-values will change less when the difference approaches 0.

Validating using SGD

After one epoch, the model learned is validated using a different data set. One epoch is after all training examples have been used for training on average, so after N examples. For validation the LCL is calculated by using the following equation:

$$LCL = \sum_{i=1}^{n} log(p_i) + \sum_{i=1}^{n} log(1 - p_i)^{[3]}$$

The left sum is for positive examples, and the right one is for negative ones. The training was finished when this value converged.

Runtime and space complexity using SGD

The SGD algorithm works on N training examples, where each of them have a d-dimensional feature vector. In addition the algorithm uses a vector of weights β which it updates for each iteration. This vector has a weight per feature in the examples. The space complexity for the SGD algorithm is $O(N \cdot d)$. In each iteration the algorithm updates d weights, and an epoch consists of N iterations. Hence, one epoch takes $O(N \cdot d)$ time. How many epochs necessary to converge on values for the different weights depend on the learning rate, strength of the regularization and the underlying model. The SGD algorithm takes $O(e \cdot N \cdot d)$ time where e is the number of epochs.

L-BFGS

L-BFGS was also used as an alternative method to SGD for learning our Beta parameters. Instead of implementing this algorithm ourselves, Mark Schmidt's minFunc was used. Regardless of option used with this algorithm, if it reached convergence the error rate observed averaged around 11%. Despite similar error rate results, this algorithm runs considerably faster($> 5x$) than the gradient descent.

Grid Search

Grid Search was used for finding the best values for the hyperparameters. This was used for the strength of the regularization for both SGD and L-BFGS, and also the learning rate for SGD. A generalized exponential scale search was first performed to find the regions of operations for each parameter, followed by a narrow scope search for more precise findings in regions of interest.

Experiments

Design of Experiments

The data set used for learning the model consisted of 559 examples. The SGD algorithm chose 500 of them at random to use for learning beta. The remaining 59 were used as a validation subset for determining the error rate. The advantage of keeping the examples used for learning and the ones used for validation independent is to avoid false belief in an arbitrarily low error rate.

In addition, because of the stochastic nature of the method, multiple trials were performed and their error rates averaged to determine results. In the results below, it is implicitly implied that an averaging operation was performed on each data point, with a minimum of 20 trials conducted.

After using this method in conjunction with a grid search to determine the optimal parameter values lambda (learning rate) and mu (regularization constant), a model for beta was derived from the training set of 559 examples. This model was tested against an independent test set of 239 observations. We attempted to classify this set using our beta values for the logistic regression model. A comparison of the classifications with the actual class values determined the final error rate.

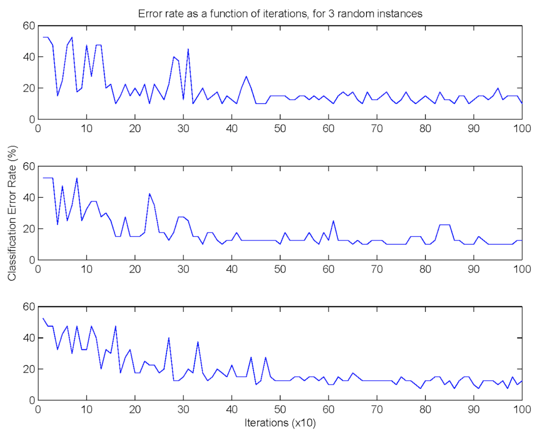

Figure 1: Error drop for 3 iterations

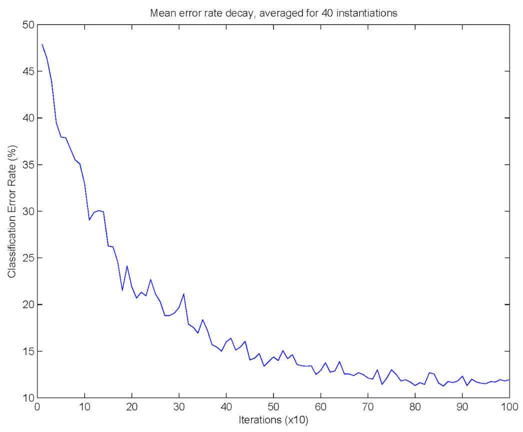

Figure 2: Averaged error drop

Results of Experiments

Figure 1 shows 3 independent tests of the performance of the model on the validation subset while the model is being learned. The randomness shown in these figures is due to the stochastic nature of the implementation. What is important to note is that, despite the sometimes backwards progress from one iteration to the next, overall progress is towards a decreasing error rate. This can be seen more clearly in figure 2, which is a plot of the average result over 40 separate tests. As the model converges towards an optimal beta vector, the error rate drops towards its final value.

An exponentially scaled grid search was performed to determine optimal hyperparameter values, μ and λ. The value of λ represents rate of gradient ascent and is linearly related to rate of convergence, but for sufficiently low values has small effect on error rate. As seen in figure 3, L values below L_0 (1 in our case) have approximately identical performance, after accounting for the randomness inherent in our experiments. However, as λ becomes sufficiently large, it is possible to 'overshoot' the optimal beta values; the gradient algorithm lacks the precision to accurately determine β. It is also interesting to note that the worst case scenario error rate is 0.5, not 1. This is due to the fact that as β loses its significance our prior class knowledge dominates the posterior probabilities derived from our feature vector.

In figure 4 a similar analysis is performed for the regularization parameter μ. As μ grows arbitrarily large, the L_2 regularization drives β towards 0, leading to the case of prior probabilities dominating. For sufficiently low values of μ, up to and including $\mu = 0$, Mu appears to have little effect on the outcome, implying that regularization is unimportant for this data set. This implies a degree of similarity between the various data points; i.e. a subset chosen at random can serves as a reasonable model of the whole.

An analysis of error as a function of our training set shows the intuitive result that a larger set leads to better results. As seen in figure 5, error rate decreases with increasing

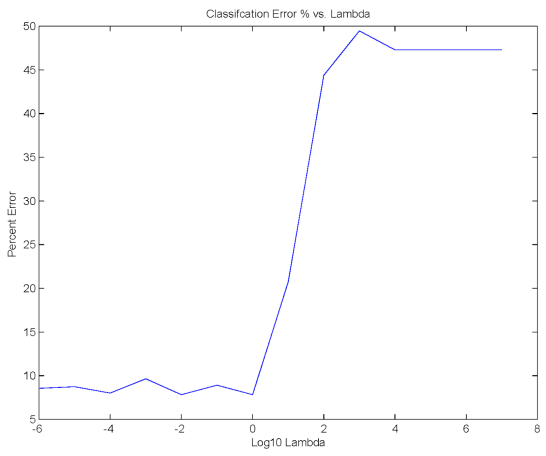

Figure 3: Error Rate vs. λ

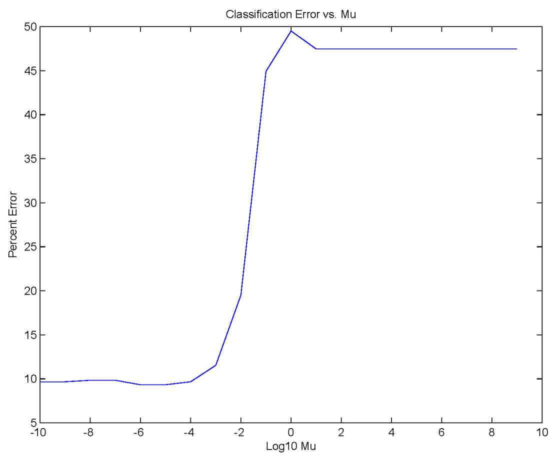

Figure 4: Error Rate vs. μ

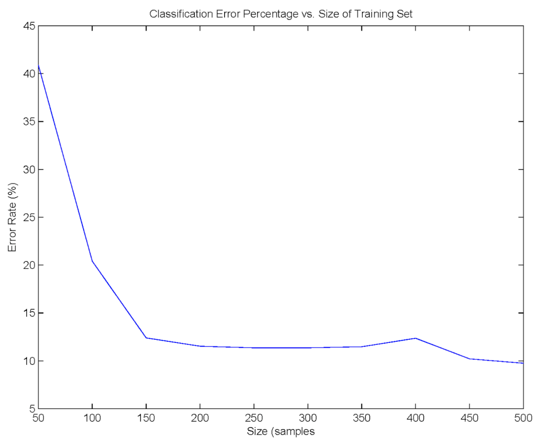

Figure 5: Error rate vs. Size of Validation set

training set, which was the motivating factor behind having a much larger training set vs. validation set.

Findings and Lessons Learned

Using the simple linear regression model we are able to model binary classification data from an unknown distribution, with accuracy rates up to an average 90%. What is significant in this is that although our algorithm assumed a simple linear model, we had no prior knowledge of the validity of such an assumption. Our results imply that, by properly using a validation subset, it is possible to apply and verify a model in poorly understood situations. Due to the simplicity of implementation and low time complexity of this algorithm, we believe it can serve as a useful first step in many machine learning situations. In determining the Beta parameters for our model We used both our own implementation of $L_2 - SGD$ and an off the shelf L-BFGS, with both leading to similar results. This implies that the model itself may be more important than the method used for parameter estimation.

In the beginning the training data was evenly split into two sets, one for training and one for validating. When the size on the set used for training was increased, and the set used for validating was shrunk, the error rate decreased. We believe this is due to the fact that as more samples are introduced, giving the algorithm more data to work with, the algorithm is able to build a more general model. In addition shrinking our validation subset lowered the computation time to determine the current error rate, which led to improved runtime.

References

1. http://www.di.ens.fr/~mschmidt/Software/minFunc.html
2. http://docs.python.org/2/library/random.html
3. http://cseweb.ucsd.edu/~elkan/250B/logreg.pdf