UNIVERSITY OF CALIFORNIA, SAN DIEGO

CSE 250B: Project 1

Sandi Calhoun,[*] Kemal Eren,[†] Alexander Xydes[‡]

January 23, 2014

We implement logistic regression using two optimization methods: Stochastic Gradient Descent and L-BFGS. This implementation was tested on the Gender Recognition dataset, achieving accuracy scores of 0.9121 and 0.9205, respectively.

1 INTRODUCTION

For this project we want to understand logistic regression, gradient-based optimizations, and the issues that arise when applying these algorithms to practical datasets. We use both Stochastic Gradient Descent (SGD) and the limited memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) algorithms to train a logistic regression model on the Gender Recognition dataset from (mlcomp.org, 2014).

Given a tuple $\langle x, y \rangle$, where $x \in \mathbb{R}^{d+1}$ and $y \in \{0,1\}$, logistic regression models the probability that $y = 1$ as:

$$p\left(Y = 1 | X = x; \beta\right) = \text{logit}^{-1}\left(x \cdot \beta\right) \tag{1.1}$$

where $\beta \in \mathbb{R}^{d+1}$, $x_0 = 1$, and β_0 is an intercept parameter. Then the log conditional likelihood of a set of independent, identically distributed samples, $\left\{\langle x_i, y_i \rangle\right\}_{i=1}^{N}$, is

* skcalhoun@ucsd.edu
† keren@ucsd.edu
‡ axydes@ucsd.edu

$$LCL = \sum_{i=1}^{n} \log f\left(y_i | x_i; \beta\right) \tag{1.2}$$

The model parameters β can be estimated by the principal of maximum likelihood:

$$\hat{\beta} = \arg\max_{\beta} LCL \tag{1.3}$$

To prevent overfitting, we used L2 regularization:

$$\hat{\beta} = \arg\max_{\beta} RLCL = \arg\max_{\beta} \left(LCL - \mu ||\beta||_2^2\right) \tag{1.4}$$

where μ controls the tradeoff between maximizing the LCL and minimizing the norm of β.

The rest of this paper is organized as follows: Section 2 describes the two algorithms we implemented to perform this optimization. Section 3 details how we tested these implementations on a dataset from mlcomp. Section 4 gives the results of that experiment, and Section 5 concludes with our thoughts on the project.

2 Design and Analysis of Algorithms

We implemented two methods to optimize the logistic regression model: Stochastic Gradient Descent (SGD) and limited memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS).

Algorithms were implemented in Python, with dependencies on NumPy, SciPy, and scikit-learn (Jones, Oliphant, Peterson, et al., 2001) (Pedregosa et al., 2011). We used the Anaconda (Analytics, 2014) distribution of Python 2.7, which includes these dependencies, to make setup simple. We wrote unit tests that verify the correctness of our implementation on synthetic data generated by scikit-learn.

2.1 Stochastic Gradient Descent

The values of β that maximize the LCL can be found by iterative gradient-following:

$$\hat{\beta}^{[t+1]} \leftarrow \hat{\beta}^{[t]} + \lambda \nabla RLCL. \tag{2.1}$$

However, each iteration achieves a small change in β at the cost of $O(Np)$ operations. Stochastic gradient descent improves this computational cost by estimating the gradient one sample at a time:

$$\hat{\beta}^{[t+1]} \leftarrow \hat{\beta}^{[t]} + \lambda \left((y_i - p_i)x_i - 2\mu\beta\right) \tag{2.2}$$

This approach is more efficient, taking $O(p)$ operations.

The hyperparameter λ controls the learning rate. After each epoch our implementation sets $\lambda \leftarrow \tau\lambda$, where $\tau \in (0,1]$, so that the step size gets smaller as $\hat{\beta}$ converges to the true solution.

Before using SGD we shuffle the samples in the training data, to ensure the samples are in random order. Then SGD iterates through them sequentially. The data is not reshuffled after each epoch.

After each epoch, the change in the regularized LCL is examined. If it differs by less than 1×10^{-8} we assume the run has converged. Execution stops if SGD has not converged after 1000 epochs.

SGD is memory efficient. For this project the whole dataset fits in memory, so our implementation requires $O(Np)$ space. However, because only β and one sample x_i need be in memory at one time, it is possible to write it so that it requires only $O(N+p)$ space.

2.2 L-BFGS

L-BFGS is an optimization method that takes as input both f and ∇f. We use the implementation of L-BFGs provided by SciPy: `scipy.optimize.fmin_l_bfgs_b` (Morales & Nocedal, 2011). All of this function's parameters are set to their defaults.

The LCL, its gradient, and the regularized versions of both are all implemented individually. They are tested using `scipy.optimize.check_grad`, which compares the gradient ∇f to a numerical estimate computed directly from f. To maximize the regularized LCL, we provide it and its gradient to `fmin_l_bfgs_b`.

2.3 Floating point precision

Both methods need to compute $p(Y=1|x,\beta) = \text{logit}^{-1}\left(x\cdot\beta\right)$. When $x\cdot B$ is large or small, $\text{logit}^{-1}(x\cdot\beta)$ will overflow to 1 or underflow to 0. To avoid this problem, we used the following equality [1]:

$$\begin{aligned}
\log(\text{logit}^{-1}(z)) &= \log\left(1/(1+\exp(-z))\right)\\
&= -\log\left(1+\exp(-z)\right)\\
&= -\log\left(\exp(0)+\exp(-z)\right)\\
&= -m-\log\left(\exp(-m)+\exp(-z-m)\right)
\end{aligned}$$

where $m=\max(0,-z)$. This ensures that the largest value passed to exp is 0, thus preventing overflow.

2.4 Analysis of Derivatives

We use `scipy.optimize.check_grad` (Jones et al., 2001) to verify that our derivatives are correct. `check_grad` takes pointers to the objective function (f), and the gradient function (∇f) and an

[1]adapted from `http://lingpipe-blog.com/2012/02/16/howprevent-overflow-underflow-logistic-regression/`

initial x value. It then computes $\nabla f(x)$ and also numerically approximates its value using f. Let $a = \nabla f(x)$ and let b be the numerical approximation of this same value; check_grad then returns:

$$\frac{||a-b||_2}{||a+b||_2} \tag{2.3}$$

which should be close to zero if the gradient function is correct.

To test our gradient functions we generated a synthetic dataset using the `make_blobs` function from scikit-learn (Pedregosa et al., 2011) with 100 samples, 3 features, and 2 classes. We generated a list of 100 β vectors, with entries drawn uniformly from $[-10, 10]$. Then we ran `check_grad` over all 100 β vectors for both the LCL and $RLCL$ functions and their corresponding gradient functions. Every run of `check_grad` returned a value less than 1×10^{-2}, so we know that our gradient functions (for both LCL and $RLCL$) work correctly.

3 Design of Experiments

We tested our implementation on the Gender Recognition dataset (mlcomp.org, 2014). This dataset is a binary classification challenge with 798 samples and 800 features. The data is seperated into a training set of 559 samples and a test set of 239 samples.

3.1 Hyperparameter grid search

Thirty percent of the training data was set aside as a validation set. Hyperparameters (learning rate λ, learning schedule τ, and regularization strength μ) were chosen by grid search. Models were trained on the remaining 70% of the training data, and the best was chosen by evaluating their performance on the validation set. Then the winning model was trained on the entire training set and evaluated on the test set.

The following values were used for the grid search:

- λ: 10^x for $x \in \{-4, -3, -2, -1, 0\}$
- τ: $\{0.3, 0.6, 0.9\}$
- μ: 10^x for $x \in \{-3, -2, -1, 0\}$

Models were compared according to the accuracy metric: (# correct)/(# samples).

3.2 Normalization of features

We compute the mean μ_j and standard deviation σ_j of each feature column vector in the training data. These parameters are used to normalize the data before training so that each feature has a mean of 0 and a standard deviation of 1.

Before prediction, these parameters are used to normalize each sample to $x_{ij} \leftarrow (x_{ij} - \mu_j)/\sigma_j$.

4 Results of Experiments

Algorithm run times were obtained on a laptop with an intel i5 processor and 16 GB of RAM.

4.1 Stochastic Gradient Descent

Our implementation of the SGD algorithm achieved an accuracy of 0.9121, with a corresponding error rate of 0.0879, for the test data set. The chosen hyperparameters were $\mu = 1$, $\lambda = 0.01$, and $\tau = 0.6$.

To ensure that this value of μ was chosen from a sufficient range, the accuracy on the validation set was determined for different values of μ (Figure 4.1).

To quantify running time, training and prediction were repeated 3 times and their running times averaged. Training converged after 40 epochs and took 2.3633 seconds total, or 0.0591 seconds per epoch. Prediction took 0.0102 seconds total, or 4.2678×10^{-5} seconds per sample.

With a run time of 2.3633 seconds, 40 epochs, 559 training examples per epoch, 800 features per example, 3 floating point operations per feature in each example, we achieved a calculation rate of 22.71 MFLOPS. On an Intel i5, with a clock rate of 2.4GHz we should be able to achieve at least a 10x speedup, possibly even a 100x speedup with more work.

After compiling the results of the grid search and choosing a value of 1 for μ, it appears that as long as the learning rate was not too small and not too large, it did not have a big impact on the accuracy of our SGD implementation (Fig. 4.2). However, the learning schedule did have an impact on the accuracy of the SGD algorithm. Decimating the learning rate a little more than halfway after every epoch gives our implementation the best results (Fig. 4.3).

4.2 L-BFGS

Our implementation of the L-BFGS algorithm achieved an accuracy score of 0.9205, with a corresponding error rate of 0.0795 for the test data set. The chosen value of μ was 0.001.

To ensure that this value of μ was chosen from a sufficient range, the accuracy on the validation set was determined for different values of μ (Figure 4.4).

To quantify running time, training and prediction were repeated 3 times and their running times averaged. Training took 1.8509 seconds. Prediction took 0.009995 seconds total, or 4.1820×10^{-5} seconds per sample.

With a run time of 1.8509 seconds, 52 function calls, 559 training examples per function call, 800 features per example, 6 floating point operations per function call, we achieved a calculation rate of 75.383 MFLOPS. On an Intel i5, with a clock rate of 2.4GHz we should be able to achieve at least a 10x speedup.

4.3 Comparison of Algorithms

L-BFGS took slightly less time to converge. Our implementation of SGD could be further optimized by rewriting performance-critical code in Cython, but since it was already fast enough for this project we did not bother to do so. L-BFGS also produced a slightly higher accuracy score (0.9205) than our implementation of SGD (0.9121) on this dataset. While L-BFGS is more efficient, this makes sense as it is a library implmentation and has had a lot more development time go into it to improve it's performance. And our SGD implementation is only half a second behind the L-BFGS library in efficiency.

5 Findings and Lessons Learned

The svmlight-linear software (mlcomp.org, 2014) reports an error rate of 0.084 for the Gender Recognition dataset. While our SGD algorithm achieved a similar error rate (0.0879), our L-BFGS algorithm achieved an error rate 0.0045 lower than that (0.0795).

Surprisingly, picking a good learning rate was less important than picking a good learning schedule for the learning rate once the regularization factor had been chosen. While the learning rate definitely can't be too large (SGD would not converge) or too small (SGD would take too much time to converge), any reasonable value will work. The learning schedule affects how quickly the learning rate is decimated, and therefore how fast the SGD algorithm moves to smaller step sizes. This affects how quickly the SGD algorithm matches the magnitude of the gradient as it gets closer to the maximum. All of this reinforces how much of an art picking the values of the hyperparameters is. The values can't be too tailored to the training or validation set, otherwise they'll lead to worse performance on future data.

While implementing the algorithms, we learned that it is important to prevent the values from overflowing or underflowing the computer's precision as that will cause unexpected behavior in the algorithms. While this might not be a issue for contrived datasets, it definitely is for a lot of real-world datasets. This will skew the results of the algorithms and make them less accurate.

References

Analytics, C. (2014, January). *Anaconda.* Retrieved from `https://store.continuum.io/cshop/anaconda/`

Jones, E., Oliphant, T., Peterson, P., et al. (2001). *SciPy: Open source scientific tools for Python.* Retrieved from `http://www.scipy.org/`

mlcomp.org. (2014, January). *Gender recognition [dct].* Retrieved from `http://mlcomp.org/datasets/1571`

Morales, J. L., & Nocedal, J. (2011, December). Remark on &ldquo;algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound constrained optimization&rdquo;. *ACM Trans. Math. Softw.*, *38*(1), 7:1–7:4. Retrieved from `http://doi.acm.org/10.1145/2049662.2049669` doi: 10.1145/2049662.2049669

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

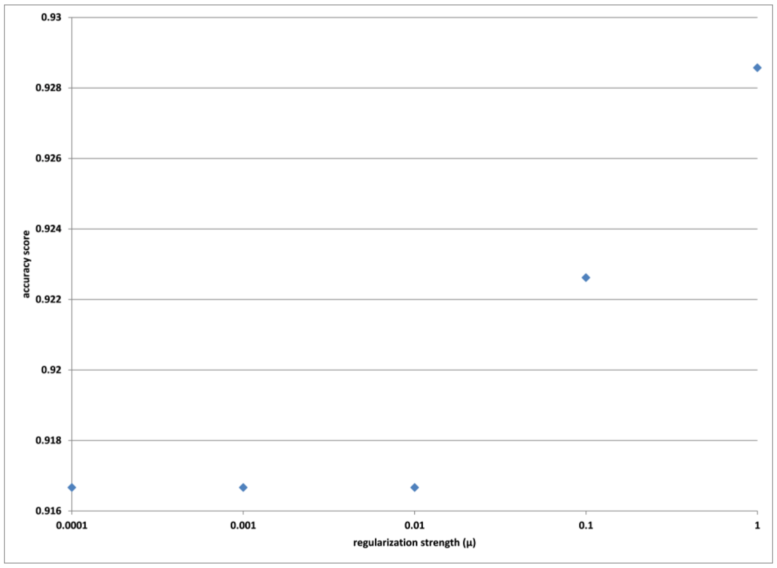

Figure 4.1: Accuracy of SGD as a function of regularization strength (μ)

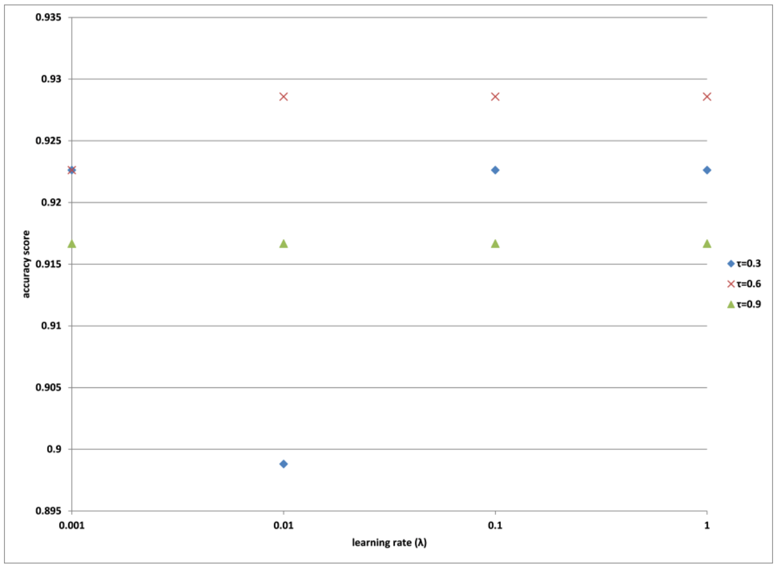

Figure 4.2: SGD: Learning rate (λ) and learning schedule (τ) vs accuracy score using a regularization factor (μ) of 1.0

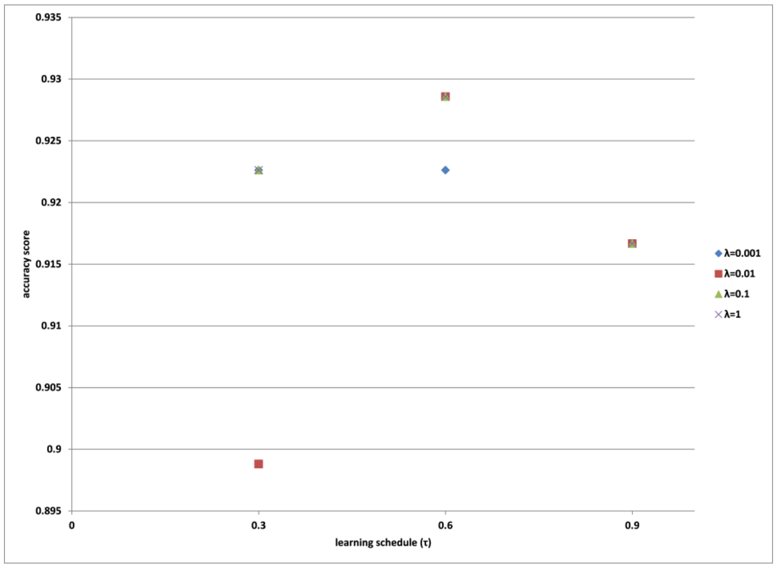

Figure 4.3: SGD: Learning schedule (τ) and learning rate (λ) vs accuracy score using a regularization factor (μ) of 1.0

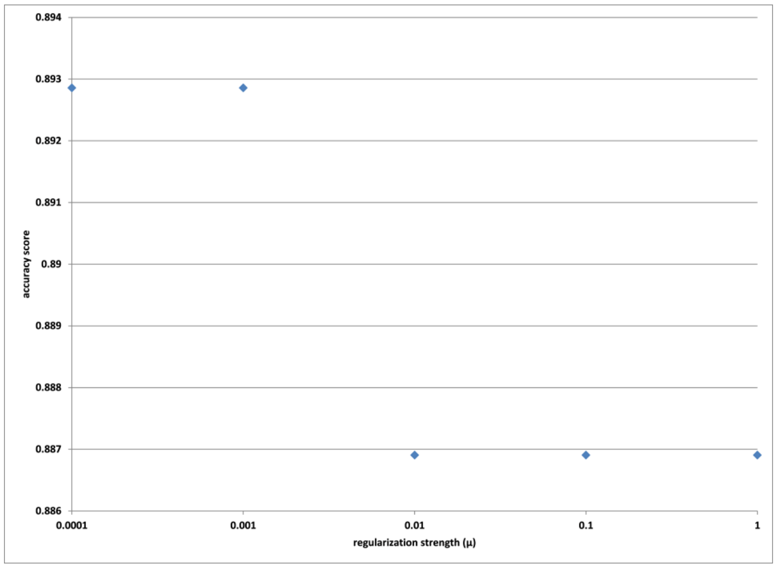

Figure 4.4: Accuracy of L-BFGS as a function of regularization strength (μ)