Reading Minds is Easy, MRI-te?

Shilpa Kumar
Department of Computer Science
University of Washington
Seattle, WA 98105
shilpak@cs.uw.edu

Siddhartha S. Gorti
Department of Computer Science
University of Washington
Seattle, WA 98105
sgorti3@cs.uw.edu

Abstract

This project involved discovering how fMRI brain scans can be used to predict what word a person is reading based off of activation patterns in the brain. The goal was to first learn 218 sparse linear models, each predicting a semantic feature of the word based on an fMRI input. Based on those 218 models, we were then able to figure out which word a brain scan was more likely corresponding.

1 Approach

Our dataset has input vectors with more than 21,000 features. A large value of D means that the dataset is very prone to over fitting when training models based on the data. One way to avoid over fitting is to generate sparse linear models, since a decrease in the magnitude of coefficients decreases the likelihood of over fitting the data. Sparse linear models can be generated using L1 penalty / Lasso regression, since coefficients can be pushed to 0.

Using Coordinate Descent and Stochastic Coordinate Descent for Lasso, as well Proximal Gradient Descent with L1 penalty, we figured out the coefficients for a sparse linear model used to predict the value of a particular semantic feature. We then proceeded to build all 218 linear models, one for each semantic feature of the words the brain scans corresponded to. We performed this both sequentially and in parallel, comparing which methods work better for a large dataset such as this one. Finally, given a new brain scan input, we were able to build a 218-dimensional vector representing the values of semantic features of the word read using the 218 models we built, and given two candidate words, we used 1-NN regression to choose the word whose semantic feature vector was closer to the predicted one.

2 Methods for Building Sparse Linear Models

We used multiple methods to find the sparse linear model for each semantic feature, to minimize over fitting. Lasso is a regression analysis method that uses $\lambda||w||_1$ to penalize the magnitude of the coefficients. Lasso allows coefficients to be 'pushed' to 0; this allows for us to achieve sparse linear models without compromising some important features of the model. Each different lasso method used was tested with multiple L1 penalty tuning parameters, and across several semantic features.

2.1 Coordinate Descent for LASSO

Coordinate descent is used to build models in cases where the gradient of the loss function cannot be easily computed. Lasso uses $\lambda||w||1$ as a penalty on the loss function, with λ being

the tuning parameter. The absolute value function is not differentiable at x = 0, and a way to work around the complicated gradient that results is to use coordinate descent instead of gradient descent.

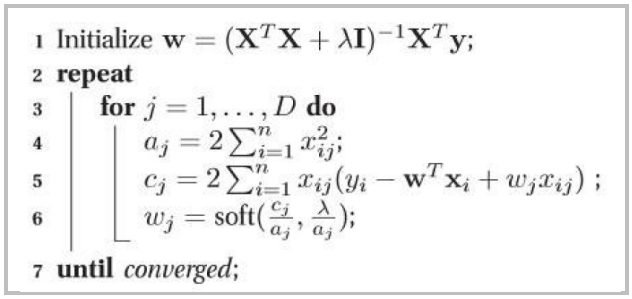


Figure 1: General Algorithm for Coordinate Descent for LASSO[1]

2.2 Stochastic Coordinate Descent (SCD)

The coordinate descent algorithm involves going through all data points multiple times per iteration, which can be time-consuming for large datasets. To combat this problem, we utilized stochastic coordinate descent, which uses one random index associated with the voxel (one column of input matrix X) per iteration to minimize the loss function L(w, λ).

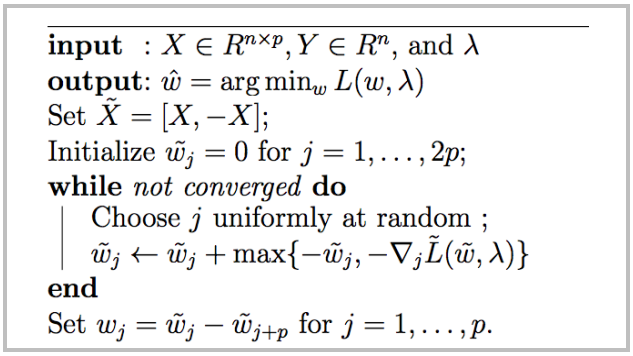


Figure 2: General Algorithm for Stochastic Coordinate Descent[2]

In order to make our method run faster, we followed optimizations outlined in the source of the algorithm to cache X˜w˜, so that we do not compute the dot product each time.

$$\tilde{L}(\tilde{w}, \lambda) = \frac{1}{2n}\|Y - \tilde{X}\tilde{w}\|^2 + \lambda \sum_{j=1}^{2p} \tilde{w}_j \quad \tilde{w}_j >= 0, j = 1, \ldots, 2p$$

Figure 3: Loss function for Stochastic Coordinate Descent

[1] Murphy
[2] Homework 3, CSE 599

$$-\nabla_j L(\tilde{w}, \lambda) = \frac{1}{n}(y - \tilde{x}\tilde{w})(-\tilde{x}_j) + \lambda$$

Figure 4: Gradient of loss function with respect to $\tilde{w}_j$

2.3 Proximal Gradient Descent (PGD)

In order to compare our previous algorithms with a method that does not use a version of coordinate descent, we used proximal gradient descent with L1 penalty. Proximal methods are useful for large-scale problems where other methods might be too slow.[3]

$$\begin{aligned}
\boldsymbol{\theta}_{k+1} &= \underset{\mathbf{z}}{\operatorname{argmin}} \left[t_k R(\mathbf{z}) + \frac{1}{2}||\mathbf{z} - \mathbf{u}_k||_2^2 \right] = \operatorname{prox}_{t_k R}(\mathbf{u}_k) \\
\mathbf{u}_k &= \boldsymbol{\theta}_k - t_k \mathbf{g}_k \\
\mathbf{g}_k &= \nabla L(\boldsymbol{\theta}_k)
\end{aligned}$$

Figure 5: General Algorithm for Proximal Gradient Descent

Since $L(\theta_k)$ is the loss function (in this case, the residual sum of squares), the gradient of $L(\theta_k)$ becomes the gradient ($\|wX - y\|^2$), which is $2X^T(wX - y) = g_k$

3 Methods for Choosing Tuning Parameters

While the goal of model-building is to minimize error over our data set, it's important to realize the distinction between a model that is capturing the true relationship versus a model that is simply over fitting the data. To help us solve this problem, we introduce the concept of regularization which helps prevent the generation of over-fit models. In building these models, we used two different methods to select our tuning parameters, k-fold cross validation and test set validation.

In addition, because the project relies on 218 separate linear models being built, simply choosing one value of the tuning parameter was not enough. Instead, for each semantic feature, the best value of the tuning parameter would have to be computed to ensure low model error.

3.1 K-Fold Cross Validation

K-fold cross validation consists of randomly splitting the data into k sections or folds and then building the model on k – 1 folds of the data set. Then, this model is tested on the kth fold and repeated for all k folds. The average of the error reported across all k folds is known as the cross-validation error. The value of the tuning parameter that minimizes this error is the regularization value selected to build the model. While this model was implemented in our project, running K-fold cross validation is an expensive operation and using smaller values of K introduces high bias into the model. Thus, we opted to use test set validation to choose our regularization parameter.

3.2 Test Set Validation

[3] Murphy

Test set validation involves setting aside some of the data as a test set. When building the model from the training set, the points from the test set are completely ignored. After the model is built error from applying the model to the test set is computed. The value of the tuning parameter that minimizes the error on the test set is the value that is selected to use to build the model for that particular feature. While this is not as comprehensive as k-fold cross validation, it is much faster, and because this computation needed to be done on every model, this is the method we used to choose the tuning parameter. It should be noted that test set validation is known to be “overly optimistic”.

4 Methods for Building All 218 Sparse Linear Models

4.1 Building Sequentially

Initially, our code was designed to build each of the 218 sparse linear models one by one. This was to make sure all of our algorithms were working, and so we could easily debug. Once we were satisfied with the quality of our code, we started thinking about how we could cut the long run time of the programming and tried to examine where we could parallelize the program.

4.2 Building in Parallel

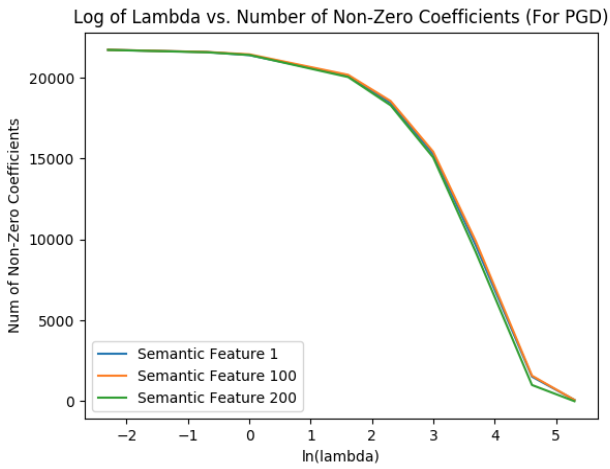
Building all 218 models in parallel also was a lot faster than building sequentially, as expected. The amount of time saved (almost a 30% speedup) was significant. This was achieved while building just two models in parallel at a time. Initially, we did encounter some problems with race conditions and possible “dirty” reads and writes. To solve this problem, we used locks to lock some critical sections of the program where we only one want one sub-process to execute. The library used for concurrent programming splits into sub-processes across different CPU cores, so a future experiment would be to see the kind of speedup that could be achieved from building 4 or 8 models at a time on a multicore processor.

5 Methods for Performing Binary Classification

5.1 1-NN Regression

1-NN regression is a non-parametric method used for classification. The algorithm picks the data point closest to the one in question, and sets the y-value we are trying to find as the y-value of the closest data point. The ultimate goal of the project was to find the best guess of what word an input brain scan corresponds to. To do this, we used 1-NN regression with two sample semantic feature vectors corresponding to two different words, and passed in the semantic feature vector we generated from an input brain scan and our 218 models. We then calculated which sample vector was closer to the vector generated by the models by minimizing the Euclidean distance, and thus set the value of the word for the generated vector to the word corresponding to the closer vector. We tested the accuracy of our 218 models by passing in the semantic feature vectors for various brain scan inputs from the test data, as well as the the semantic feature vectors for the actual word the brain scan corresponded to and a random different word.

6 Graphs of Results



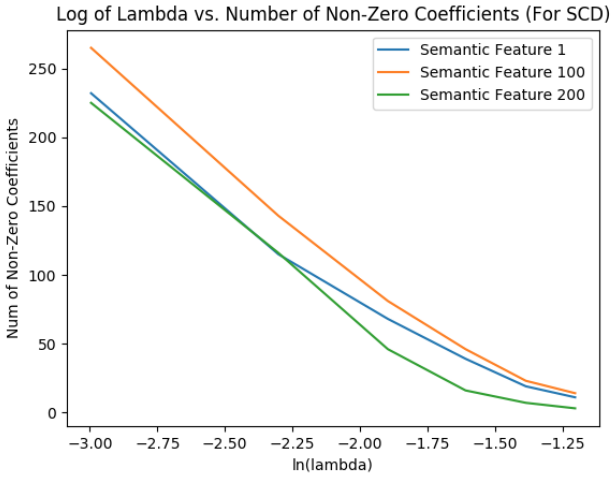


Figure 6: Log of tuning parameter vs number of nonzero coefficients

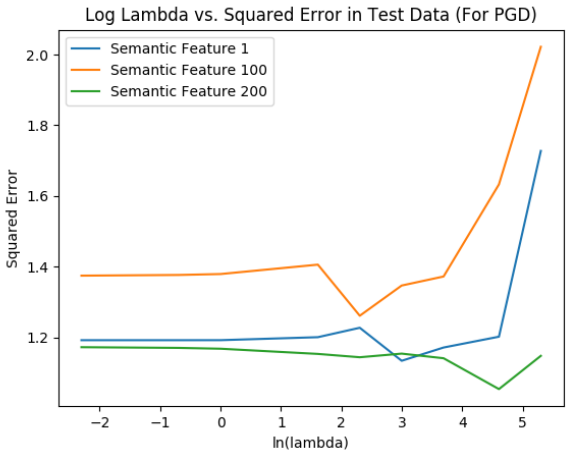


Figure 7: Log of tuning parameter vs test and training error for PGD

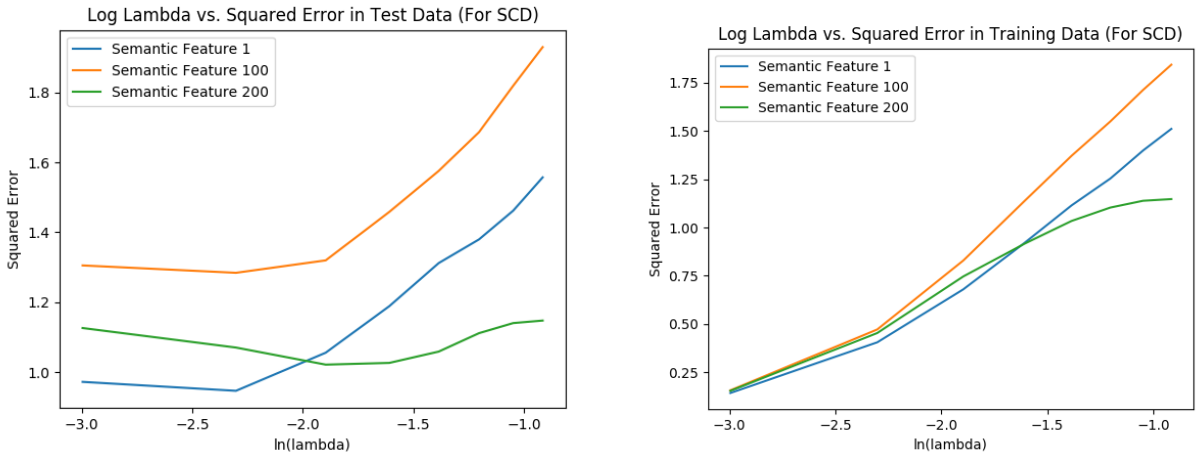


Figure 8: Log of tuning parameter vs test and training error for SCD

Table 1: Time taken to build 218 models

Method	**Time Taken (minutes)**
Sequential	84.802
Parallel	58.130

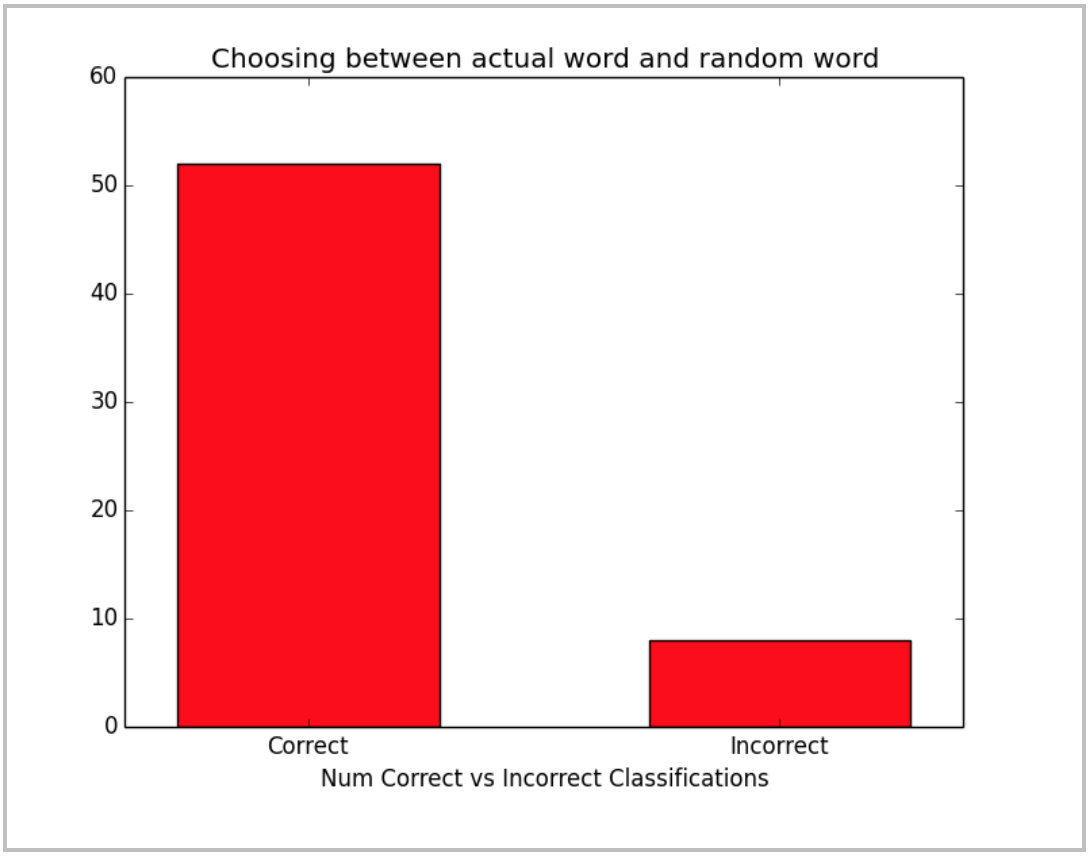


Figure 9: Number of correct versus incorrect classifications using 218 SCD models

7 Analysis of Results

Coordinate Descent for Lasso, on our dataset, took an unreasonably long amount of time. For predicting the model of just one semantic feature, the algorithm took upwards of 2.5 hours. We used coordinate descent simply as a baseline comparison, but did not calculate training error or test error for various lambdas because of the length of time necessary for 218 models and various lambdas.

Both our stochastic coordinate descent (SCD) and proximal gradient descent (PGD) followed the correct trends in training error and test error. Training error exponentially increased for various semantic features when lambda was increased, and test error decreased up to a certain lambda, and increased again. However, the graphs for PGD seemed less smooth than those of SCD, and the training error graph for PGD had a small dip in error near $\ln(\lambda) = 2$. In addition, PGD took more time to complete than SCD for the same number of iterations. For the number of nonzero coefficients vs λ, convergence took much longer for PGD. Thus, we decided to use SCD to build all 218 sparse linear models.

For all three methods used to build a semantic feature model, we found that λ values need to be much lower than typical λ values we experimented with in class.

We generated all 218 sparse linear models using SCD both in a sequential and parallel manner. We found that the parallel method helped build those models in 54 minutes, while doing it sequentially took around 85 minutes. Although it was expected that parallel would reduce the computation time, we did not expect to reduce by so much time. This shows that making optimizations such as building the models in parallel makes a huge impact when working with such large datasets.

Since the test dataset had 60 brain scan input vectors, one for each unique word, we tested our 1-NN classification, as well as the accuracy of our models at predicting the correct word,

using the test data. We used the 218 models generated by our SCD algorithm to build a 218-dimensional vector for each brain scan input, representing the semantic feature vector as predicted by our algorithm. Then, we proceeded to compare the generated semantic feature vector (x_q) with two semantic feature vectors x1 and x2, with x1being the vector for the correct word representing the input brain scan. We used 1-NN regression with x_q, x_1, and x_2 to figure out which vector was more like our generated vector, and used that to predict the word based on the input brain scan. As our graph shows, our 218 models helped predict the correct word more than 50/60 times, showing that our SCD was building models for the different semantic features pretty accurately.

8 Conclusion

Overall, we were able to learn a lot about performing linear regression and classification on very large datasets. Our results showed us that using simple coordinate descent with lasso regression was simply infeasible for a dataset so large given our computing power and time. We also noticed that SCD provided much more accurate and trustworthy results than PGD, with less fluctuation and random spikes in the graphs. SCD could perhaps be better at building models with input data that has large number of features.

Initially, we wanted to use K-fold CV so that our models would be built without "touching" the test data and so that our models wouldn't be biased when used against the test words. However, K-fold cross validation was a lot slower than test set validation, and although this was expected, the extent to which K-fold CV was slower was probably due to the magnitude of the input dataset we were using to build the models and test different λ's with.

In conclusion, it appears that the models built around the 218 semantic features were in fact good predictors of word. Even when the model was tested against similar words, celery and carrot, the model was able to correctly predict the right word spoken and out of all 60 test words given, the model had around 84% accuracy. This suggests that there is strong linkage between reading a word and the neural activation associated with thinking about word meanings. In the future, we would like to explore the effects of k-fold cross validation and the potential of using standard coordinate descent for LASSO with more computing power and time to generate different, possibly more accurate models.

References

[1] Bradley, Joseph K., Aapo Kyrola, Danny Bickson, and Carlos Geustrin. *Parallel Coordinate Descent for L1-Regularized Loss Minimization*. 2011. MS. Carnegie Mellon University, Pittsburgh.

[2] Caramanis, and Sanghavi. "Lecture 4 — September 11." EE 381V: Large Scale Optimization. UTexas. Lecture.

[3] "Homework 3 -- Midterm." CSE 599 / Stat 592: Machine Learning (Statistics) for Big Data. University of Washington Computer Science. Homework Assignment.

[4] Murphy, Kevin P. Machine Learning: A Probabilistic Perspective. Cambridge: The MIT Press, 2012. Print.

[5] Shalev-Schwartz, Shai, and Ambuj Tewari. *Stochastic Methods for L1 Regularized Loss Minimization*. 2011. MS. Toyota Technological Institute at Chicago, Chicago.