CS 181 Practical 1

Nicholas Larus-Stone and Matthew Goldberg
Team Team Goldberg

February 13, 2015

1 Ridge Regression

The first method we tried upon downloading the data was simple, ordinary least squares with no regularization. We implemented this regression method using numpy.linalg.solve to solve for the w that minimizes

$$E(w) = \frac{1}{2}(t - \Phi w)^T(t - \Phi w)$$

which is equivalent to solving the equation

$$\Phi^T \Phi w = \Phi^T t$$

However, this method failed quickly, resulting in a singular matrix; this occurred because (as we later found out) many of the columns in the original design matrix consisted only of zeroes. To fix the singularity problem, we added a regularization term to our loss function that penalizes proportionally to the sum of the squares of the weights; this turns our ordinary least squares approach into a ridge regression approach. Like we did for OLS, we implemented ridge regression using numpy.linalg.solve to find the w that minimizes

$$E(w) = \frac{1}{2}(t - \Phi w)^T(t - \Phi w) + \frac{\lambda}{2} w^T w$$

which is equivalent to solving the equation

$$(\Phi^T \Phi + \lambda I)w = \Phi^T t$$

We began our ridge regression by arbitrarily setting $\lambda = 1$. Once we had ridge regression successfully implemented, we fit it using all of the training data, used the resulting w to predict gaps for the test data, and made our first submission to Kaggle, just to see where this approach stands. This gave us a RMSE of 0.2985, which was on par with the standard ridge regression benchmark on Kaggle (as one would expect).

Once we had implemented ridge regression, we had to decide on the best value for λ. To do so, we used cross-validation techniques to determine the best value for λ based on which values of λ performed best on new, unseen data; we measured performance using RMSE, justifications for which you will find in section 3 below. For more details on how we used cross-validation to find the best λ, see section 2 below.

We liked the ridge regularization approach over other types of regularization (such as lasso) for a few reasons. First, it is relatively inexpensive from a computation perspective because there is a closed-form solution for w. On the other hand, solving lasso regularization has no closed-form solution and thus requires an implementation of gradient descent, which is more expensive because it requires repeated iteration to reach the minimum. Moreover, we prefer ridge regularization because it will penalize larger errors proportionally more; that is, ridge regularization would rather have three predictions that are off by one than one prediction that is off by three, whereas lasso regularization sees these situations as equally preferable. We thus prefer ridge regularization because consistent, small errors result in smaller RMSE than a mix of very small and very large errors, and minimizing RMSE is the ultimate goal of this competition.

2 Cross-Validation

Once we had implemented ridge regression, we had to decide on the best value for λ. To do this, we implemented (on our own) the cross-validation approach introduced both in section and in Bishop; this involved splitting the data into S folds, training each model in the comparison S times (where each run has a different fold as the validation set and uses the rest of the data for training), and finding the RMSE of the model over all S runs. The value of λ for the model that results in the smallest RMSE is then deemed the best value of λ of all those tested.

We tested a few different values of S, keeping in mind that a higher S means more runs and training on more different subsets of the data, but it also means longer runtime and more computation. In the end, we decided on $S = 5$, which we found to be the ideal blend between speed and accuracy.

We found our optimal value of $\lambda = 0.21$ by first testing values of λ that were relatively spread out and then focusing in on the range that minimizes λ. For example, we first tested eight models with λ's equally spaced between 0.05 and 1. When we found that the optimal value λ lay between 0.186 and 0.321, we looked at eight new models with λ's equally spaced between those two values. We continued this process of narrowing in on the optimal λ until the gain in RMSE was negligible.

While optimizing for λ did prove helpful and did improve our Kaggle RMSE slightly, the overall gain was not very significant, and it seemed that there was not much more we could do to improve our RMSE further after optimizing for λ. To see how we proceeded, see section 4 below on feature engineering.

3 RMSE

We calculated RMSE using the formula

$$RMSE = \sqrt{2E(w)/N}$$

where N is the size of the data set (the number of points in the data). Whenever we needed to evaluate the performance of a model throughout our work on the practical, we used RMSE. This makes sense for a few reasons. First, it penalizes larger errors proportionally more, so smaller RMSE implies that the model's predictions aren't off by much. Second, it makes sense to use RMSE because that is how our models were to be evaluated on Kaggle, so using RMSE to evaluate performance was consistent with how they would eventually be evaluated on Kaggle.

We used RMSE in a few different contexts throughout the practical. For example, RMSE was important for model evaluation and selection when performing cross-validation. Moreover, when testing a new model, we usually calculated its RMSE on the training data. While it is generally not ideal to evaluate a model based on the data it was trained on, this practice gave us a good idea of a model's efficacy without submitting to Kaggle; in fact, the RMSE on the training data was typically within about 0.01 of our Kaggle RMSE. It is beneficial to limit submissions to Kaggle for two reasons: first, because we were limited to four submissions per day, and second, because generating predictions for the entire data set in test.csv required a great deal of time. Thus, it was better for us to look at how the model was doing on the training data before we decided it was worth submitting (in which case, we would generate predictions for test.csv and submit).

4 Feature Engineering

As you have seen above, we seemed to have hit a score that we thought we could not beat using techniques learned in class. However, upon taking a closer look at the data we were given, we realized every feature save 30 was composed of all 0's! Once we understood that much of the given data was not helping us, it became clear that our next steps were to attack the problem through feature manipulation.

Luckily for us, Professor Adams posted on Piazza how he arrived at the feature list he was given. The original feature list was calculated using rdkit to create a Morgan Fingerprint with a radius of 1 and 256 features. After we read the documentation, we saw that the Morgan Fingerprint was a bit vector that indicated whether or not a molecule had certain features. If two molecules had similar bit vectors, they are likely to have similar structures, and therefore similar HOMO-LUMO gaps. Regardless of our learning algorithm, our end goal was going to be to learn which features were important in determining the HOMO-LUMO gap.

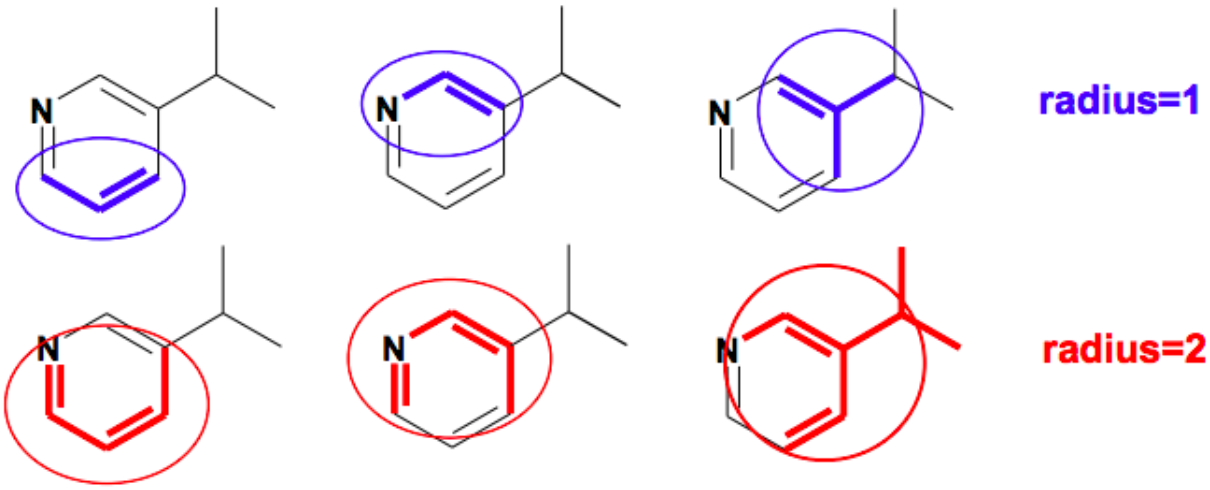

Figure 1: Image showing the difference in number of atoms reached by an increase in radius. Taken from http://rdkit.org/UGM/2012/Landrum_RDKit_UGM.Fingerprints.Final.pptx.pdf

Only having 30 non-zero features was clearly handicapping us, so we tried to gain access to more relevant features in two ways. First, we understood the radius represented the length of path searched at each atom (Fig. 1), so we looked to increase the radius. A radius of 1 means that only immediate neighbors are searched, while a radius of 2 involves searching immediate neighbors and the neighbors of immediate neighbors. So, having a limited radius is a little bit like having a flashlight in a dark forest–you can only see what is directly around you. In order to more accurately compare molecules, though, we wanted a floodlight, so we increased the radius to 2 and then 3. We were wary of increasing our radius by too much, however, because it takes longer to generate the fingerprints for larger radii as some atoms are repeated. The literature we looked at also suggested using a radius of 3, likely to avoid the problem detailed above[1][2].

Increasing the radius helped; we dropped our RMSE from 0.2985 to 0.2206 just by increasing the radius from 1 to 3. But, we want to improve on that, so we decided to increase our access to relevant features by having more features overall. One of the parameters in the GetMorganFingerprintAsABitVect is the number of bits to represent the features, so we tried increasing it to 512 and then 1024. Increasing it to 512 gave us a RMSE of 0.1822, while 1024 bits gave us a RMSE of 0.1563. However, we got these RMSEs from training and testing on just 50,000 molecules.

The way we calculated these Morgan Fingerprints required us to extract all the SMILE strings from the data, convert each one into a molecule, then run GetMorganFingerprintAsABitVect on every molecule. This is computationally expensive, as we had to do this on both the training data and the test data–potentially 1.8 million molecules. Even as we were looking at ways to reduce our RMSE, we had to be mindful of our processing power limitations.

So, when we tried to run our Ridge Regression on the full training and test sets using 1024 bits, it simply took too long. We were unable to even read in the entire training data and extract the features we wanted within a couple of hours. This meant we had to make some modifications to our approach. In the end, we decided to use fewer bits and train on less data. Our final run that we ended up submitting to Kaggle involved training on 250,000 molecules and using 512 bits for features because those were the parameters that seemed to give us the lowest RMSE while also letting us calculate the predictions without more processing power.

If we had more time, we would have also tried to add some more targeted features of our own. Based on our knowledge of HOMO and LUMO, there are a number of features that rdkit would let us extract from a molecule that could be very informative about the HOMO-LUMO gap. We would have liked to look at formal charge, numbers of rings, number of donor hydrogens and others in order to create a more precisely focused list of features. However, one of the great things about machine learning is that we don't have to know which features are directly affecting our target as long as we have enough data so that we can create a model for it.

[1] http://www.jcheminf.com/content/3/1/51

[2] http://onlinelibrary.wiley.com/doi/10.1002/anie.201310864/

5 Random Forest and Next Steps

As explained above, we soon realized that we couldn't improve our ridge regression. In class, we haven't learned any regression techniques other than least squares, ridge, and lasso regression, but it seemed like lasso regression would have similar limitations to ridge regression. While we were tackling the problem of extracting features, we were also researching alternative regression techniques. One technique that seems to be quite popular and drew our attention is the random forest learning technique, as implemented by RandomForestClassifier in scikit-learn.

We liked random forests for a number of reasons. Firstly, random forests are able to rank feature relevance, which was our primary aim when attempting to learn from the training set. This means there is a naturally fit for random forests to be able to solve this type of regression problem because the random forest would be able to learn what features are most useful, then use that information to predict the targets of the test data set.

Another reason we liked random forests was because of its implementation of bootstrap aggregating. As mentioned above, we had some worries that our ridge regression model would overfit the data due to its sparsity, so using random forests would allow us to average over a number of models. Random forests are guaranteed not to overfit the data[3].

Once we had manipulated our features into appropriate bit vectors, we started to prepare an implementation of a random forest for our data. Although we were unable to successfully implement a random forest on our entire data set, we trained it on 50,000 molecules and it gave us a RMSE of 0.1032. This was significantly better than simple ridge regression, but it also took much, much longer to run. This meant we were unable to run the random forest regression on the entire data set, but it would have likely given us a much lower RMSE if we had been able to successfully run it.

Given more time, we would have certainly finished the implementation of the random forest. One thing that we would have liked to do is tune the number of trees in our forest. Looking around at literature online, we discovered that a common number of trees is about 30, so that's what we were using to test our data[4]. However, we are aware that tuning the number of trees can have a big impact on performance and RMSE, so we would definitely have fiddled with that number to discover the optimal number of trees.

We would have also liked to explore other techniques such as neural nets and deep learning because they offer alternative methods to solving this type of problem. We look forward to encountering and using these methods later on in the class.

[3] `http://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm`

[4] `http://www.researchgate.net/post/How_to_determine_the_number_of_trees_to_be_generated_in_Random_Forest_algorithm`