Steven Tanner
CS 471
Final Project Report
12/10/20
Github - stentann

Introduction

This report goes over my work for creating machine learning models for two different datasets. The first dataset is a regression problem with 27,930 samples, each of which have 9,491 features. Each sample is a different chemical, and features describe the important physical properties of those chemicals. The labels are binding affinity to a specific unknown molecule. The second dataset poses a classification problem and has 463,715 samples, each of which have 90 features. There are 3 labels: 0, 1 and 2. Each label is 8%, 32% and 60% of the training data respectively. The model for this dataset will be evaluated equally for each label's error, so the imbalanced dataset is an obstacle. Each data point in this dataset is a song, with each feature representing how much power there is in different frequencies throughout the song. The only information I have about the labels is that they have something to do with the year the song was released.

Dataset 1 - Neural Network

Methods

The first thing I did with this dataset is look over it. The features of this dataset look sparse with many of the values being zero. The labels are floats that vary from 4.3003 to 8.0901.

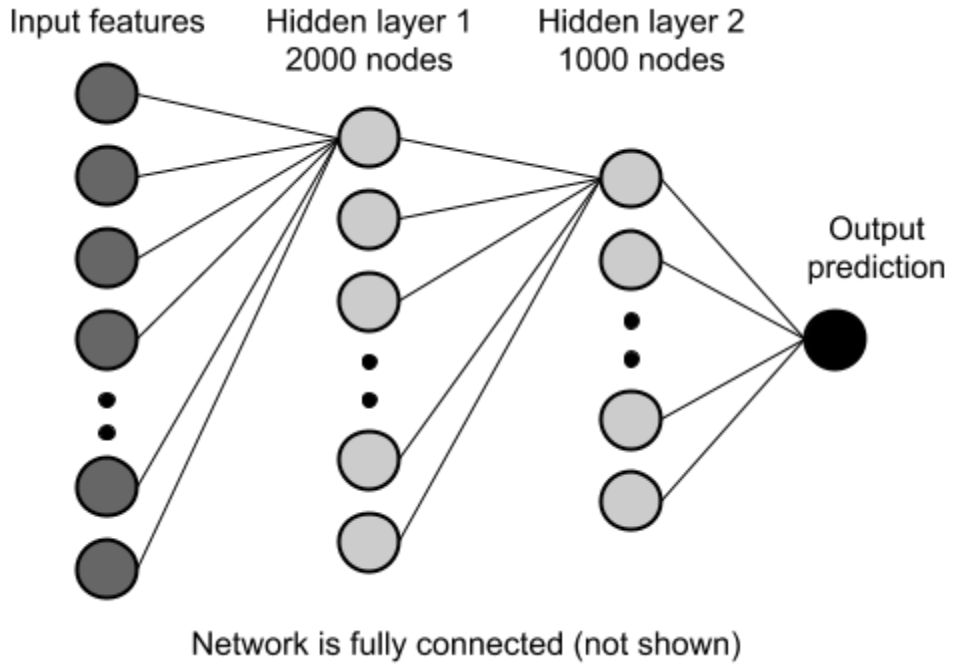
Then I ran two prediction algorithms with the purpose of comparing them to my final models to gauge performance. The first one guesses the mean of the training labels from an 80/20 train test split. This had a 0.482 mean absolute error and a 0.390 median absolute error for the test set. The other algorithm used for comparison is linear regression. I was informed by Dr. Harris, who provided me this dataset, that this problem would be better modeled by a non-linear model so this was not considered an option for the final model. After analyzing the results of the following models, it is clear to see that he was right that it can be better modeled by a non-linear model. The linear model achieved a median absolute error of 0.274 on the test set. After multiple trials of linear regression, I had incredibly large mean absolute errors on the test set: 107 million and 350 million. Given the labels fall within a range of 3.7898, this is very extreme. Most estimates from the linear regression model fall within the label range, but some of them were as far out as -1 trillion. I think that since there are roughly 3 times as many data points as there are features, and since the features are so sparse, certain combinations of features are not found in the training set which causes overfitting with large coefficients. These coefficients all work with the training set, which achieved 0.255 mean absolute error, but when certain new combinations are found in the test set, they are predicted terribly.

Given the performance and issues with linear regression, there are two non-linear models I plan to try for this problem: a neural network and a random forest. Since a neural network passes over the dataset many times and only adjusts the weights and biases slightly with each one, I

don't think it will have the same issue with overfitting that linear regression had. In fact, later in this paper I show that this is the case, given that my neural network performs well with mean absolute error in both the training and test sets. I want to try random forest because I think it will work well with a sparse dataset given that it splits the data along a line for each decision it makes. Since random forest sets specific values to guess in each leaf of a decision tree, I don't think a well trained decision forest will make guesses far outside the label range like linear regression did.

Neural Networks Explained

A neural network is a network of nodes arranged in layers. Each node in the network multiplies each output from the nodes in the previous layer by a weight and adds those values together as its output. Here is a diagram displaying the structure of my network.



Computation of each node during prediction (forward propagation):

$$p_j(t) = \sum^{i} o_i(t) * w_{ij} + w_{0j}$$

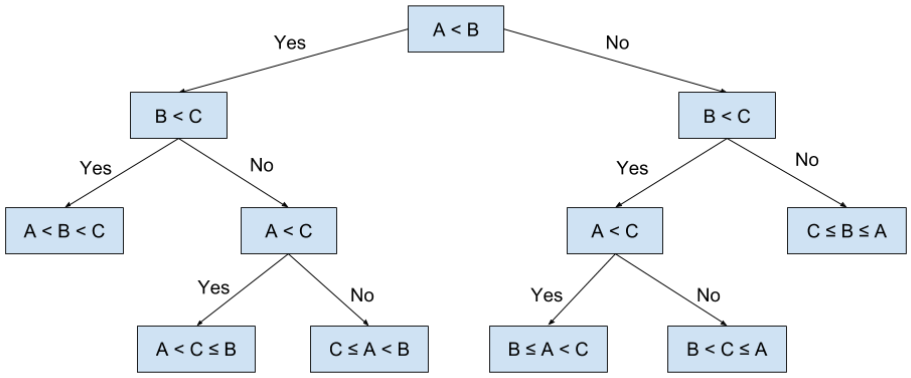
$p_j(t)$ is the output of node j, $o_i(t)$ is the output of node i from the previous layer, w_{ij} represents a weight, and w_{0j} represents a bias which is a value that isn't dependant on the previous layer

There are many different loss functions for neural networks, but I used L1 loss since my model's performance is being evaluated on mean absolute error and/or median absolute error. Here is the cost function equation:

$$Cost = 1/N * \sum^{N} \left| y_i - prediction \right|$$

I used stochastic gradient descent with mini-batching to speed up processing time. The way this works is the neural network updates its weights by using the chain rule to calculate the gradient of each weight through backpropagation. The gradient for each weight is calculated by adding the computed gradient for that weight for each data point in the batch and multiplying by the learning rate. All the weights in the network are updated after every batch using this gradient.

Random Forest Explained



The random forest model is an ensemble machine learning model that uses multiple decision trees to decide its prediction. Each decision tree is made up of decision points in each node of the tree, each with two pathways branching off from it. Each decision point asks a question about one feature of the data e.g. is feature 1 less than 0.5? The answer to these questions decides the path down the tree until it reaches a leaf. Each leaf contains the prediction that tree makes when the leaf is reached.

For training my random forest, I chose mean absolute error since my model's performance is measured by mean absolute error and median absolute error. The decisions in each node are decided by choosing the split that has the most increase in the mean absolute error while working around the other hyperparameters I will be explaining later.

$$mean\ absolute\ error\ =\ 1/N\ \sum^{N}\left|y_i - prediction\right|$$

Pre-processing

The only pre-processing I performed for this dataset is normalization. I used normalization to ensure the input data was well scaled for better performance.

$$X_{normalized} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

My Models - all error results included in a table after the analysis

Neural Network 1

- 80/20 train test split
- Mean absolute error loss criteria (also known as L1 loss)
- Stochastic gradient descent
- Learning rate 0.001

- Momentum 0.9
- Mini batching batch size 50
- ~300 epochs completed
- 2 hidden layers with 2000 and 1000 nodes

Neural Network 1 Analysis

After running through a couple iterations of this network with different learning rates and momentum values, I noticed that these values were training the network at a good speed and slowly converging on a better solution over time. I chose a batch size of 50 because it was close to maximizing my GPU's VRAM. After training for a couple hours I stopped the training at around 300 epochs because I didn't think it was reaching a good loss, this was a mistake because I was accidentally comparing the median absolute error of the linear regression to the mean absolute error of the neural network. It was also clear that training all 3000 epochs I had set it to would take over 12 hours which would not leave me enough time to run multiple models for hyperparameter tuning. Since I thought it was performing badly and taking too long to train, I switched to training random forest models.

Random Forest 1

- 80/20 train test split
- 20 trees
- Mean absolute error criterion
- Maximum features considered for each split = $\sqrt{number\ of\ samples}$
- Maximum number of samples for each tree = 100% of data points

Random Forest 1 Analysis

This model performed far better in the training set than the test set, so it was a clear case of overfitting. To solve this, for the next model I set a maximum tree depth of 6, and required 10% of the data to be in a node for the model to add a split. This is expected to help with overfitting since model depth and low numbers of samples used for a split lead to the trees fitting their decisions around small amounts of data. I also changed the train test split to 90/10 since there are 463,715 samples, which still leaves me with a test size of 46,372 samples, which is plenty.

Random Forest 2

- 90/10 train test split
- 10 trees
- Mean absolute error criterion
- Maximum features considered for each split = $\sqrt{number\ of\ samples}$
- Maximum number of samples for each tree = 100% of data points
- Maximum tree depth = 6
- Minimum number of samples in order to split = 10%

Random Forest 2 Analysis

My training and testing error were very close to each other, so I knew I wasn't overfitting anymore, but my model was still underperforming relative to the neural network, and had a worse mean absolute test error compared to the last random forest. My overfitting problem was

solved, but now I was underfitting. To fix this, I increased the maximum depth to 10, and lowered the minimum number of samples for a split to 5%. I also increased the number of trees to 30 to increase performance since I could afford some more compute time, and I turned on bootstrapping with 30% of the data being included in each tree to decrease the compute time with little risk of lowering model performance.

Random Forest 3

- 90/10 train test split
- 30 trees
- Mean absolute error criterion
- Maximum features considered for each split = $\sqrt{number\ of\ samples}$
- Maximum number of samples for each tree = 30% of data points
- Maximum tree depth = 10
- Minimum number of samples in order to split = 5%

Random Forest 3 Analysis

Unfortunately this did not increase by test performance by much. I decided to increase the maximum depth t0 30, increase maximum samples in each tree to 50% and add 20 more trees in hopes of finding some middle ground between overfitting and underfitting.

Random Forest 4

- 90/10 train test split
- 50 trees
- Mean absolute error criterion
- Maximum features considered for each split = $\sqrt{number\ of\ samples}$
- Maximum number of samples for each tree = 50% of data points
- Maximum tree depth = 30
- Minimum number of samples in order to split = 5%

Random Forest 4 Analysis

My test error metrics only got worse after increasing my model complexity. This told me that I was unlikely to find much more success than this by tweaking my model parameters. At this point I realized that I had wrongly judged my neural network, and it had performed better than anything else, so I decided to return to that model. Unfortunately I did not save my neural network's predictions because it hadn't finished the overly ambitious 3000 epochs I set it to. This means I would have to run it again from scratch. After fiddling with my model size and batch size, I found out that my computer couldn't handle a larger network, but I could increase the batch size to 100, which would cut down a bit on compute time. I changed the momentum to 0.8 since I thought 0.9 might be a bit too high (1.0 being the maximum momentum allowed), but didn't change the learning rate because it had performed well last time I ran it. I didn't have much time left for hyperparameter tuning so I stuck close to something I knew would perform decently well.

Neural Network 2

- 80/20 train test split
- Mean absolute error loss criteria (also known as L1 loss)

- Stochastic gradient descent
- Learning rate 0.001
- Momentum 0.8
- Mini batching batch size 100
- 400 epochs
- 2 hidden layers with 2000 and 1000 nodes

Neural Network 2 Analysis

This network performed slightly worse than the previous neural network in terms of mean absolute error, the only metric I measured the first network with. I think this is because I lowered the momentum which most likely helped the network converge on a good solution faster. This network outperformed the random forest models in terms of mean absolute error, which was my main method of evaluation, so I'm glad I went back to the neural network.

Results

Method	Guess the mean	Linear Regression	Neural Network	RF1	RF2	RF3	RF4	NN2
Training Error (Mean)	0.484	0.255	0.300	0.134	0.35	0.360	0.344	0.311
Training Error (Median)	0.388	0.182	Not computed	0.087	0.077	0.092	0.118	0.169
Test Error (Mean)	0.482	107 million	0.300	0.315	0.371	0.349	0.357	0.311
Test Error (Median)	0.390	0.274	Not computed	0.229	0.077	0.088	0.136	0.178

Dataset 2 - Random Forest

Methods

Oversampling Explained

This dataset is fairly imbalanced with each class consisting of 8%, 32% and 60% of the data, so I addressed this issue with oversampling. Oversampling is a method that consists of duplicating the minority classes in the training data in order to get better accuracy with regards to all classes.

Gini vs Entropy

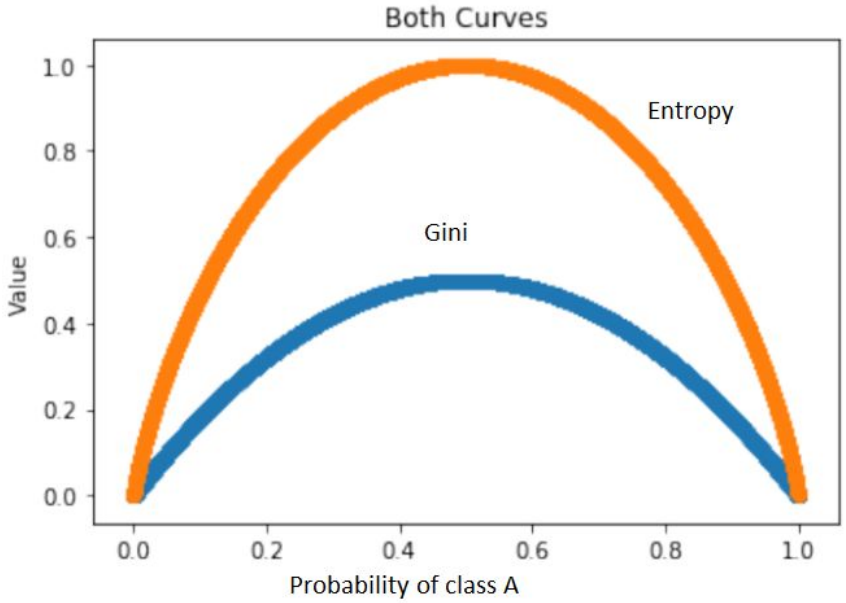
Gini and entropy are two different criteria for choosing the splits in a random forest classification model. Gini measures how often a randomly chosen element from the set would be incorrectly labeled. The formal definition is

$Gini = 1 - \sum_{j}^{j} p_j^2$ where p_j represent the probability that a specific class, j is incorrectly labeled.

Entropy is defined as

$Entropy = 1 - \sum_{j}^{j} p_j log_2 p_j$

The formulas for both methods are similar, and they accomplish the same goals, but there are some advantages and disadvantages to each. Entropy can be more useful in very particular circumstances due to its additive property. This property is unlikely to help much, so I decided to go with gini since it is faster to compute, which would let me increase other helpful hyperparameters like the number of trees.



Since this problem is a classification task, I used logistic regression to compare my non-linear models to a baseline. I will also be using the mean absolute error during hyperparameter tuning since the median absolute error does not vary much, and only takes on whole number values: 0, 1 and 2 in my performance measurements. I also used 8 different metrics for performance evaluation. For both the training and test set I created an oversampled version of it, resulting in 4 sets, each of which is measured in terms of mean absolute error and median absolute error. By oversampling the test and training sets, I can see the errors without the imbalanced data playing a factor. Since my final model is being evaluated on a balanced dataset, I will be putting more weight on the oversampled error rates when choosing hyperparameters. For this task, I only made random forest models. This is because random forests are great for imbalanced datasets due to how they split data on certain features in order to separate the different groups. Although random forests alone help with imbalanced datasets, I included oversampling to make sure the models did not exclude the small 8% class, which could easily be overshadowed by the much larger classes, even with a random forest model.

My Models - all error results included in a table after the analysis

Logistic Regression

- 90/10 train test split

Logistic Regression Analysis

It is fairly obvious that this model overfit based on the fact that the training and testing errors are so different, and all my non-linear models outperformed it for mean absolute test error on both the randomly sampled and oversampled test sets.

Random Forest 1

- 90/10 train test split
- 15 trees
- Gini criterion
- Minimum number of samples required to create a split = 10%
- Maximum number of samples in each tree = 50%
- Maximum number of features considered when making a split = 50%
- Maximum tree depth = 5

Random Forest 1 Analysis

This model outperformed the logistic regression. There was no sign of much overfitting other than a slightly worse oversampled mean absolute test error than the oversampled mean absolute train error. I figured I would increase the model's complexity in order to try to increase the performance. This model trained fast, so I added 85 trees, lowered the minimum number of samples required for a split to 5% and slightly increased the tree depth to 7 to avoid overfitting too much with a large depth.

Random Forest 2

- 90/10 train test split
- 100 trees
- Gini criterion
- Minimum number of samples required to create a split = 5%
- Maximum number of samples in each tree = 50%
- Maximum number of features considered when making a split = 50%
- Maximum tree depth = 7

Random Forest 2 Analysis

This model performed almost exactly as well as the first random forest. I decided to reduce the number of trees to 50, since that hyperparameter did not help much, and instead increase the maximum number of features checked before splitting to 100%, increasing max depth to 12, and allowing the trees to split with as little as 2 samples in a node.

Random Forest 3

- 90/10 train test split
- 50 trees
- Gini criterion
- Minimum number of samples required to create a split = 2
- Maximum number of samples in each tree = 50%

- Maximum number of features considered when making a split = 100%
- Maximum tree depth = 12

Random Forest 3 Analysis

This model overfit, but had better training loss. I think it is due to the tiny number of samples required to split a node, 2, which is very low considering the number of total data points is 463,715. It also had a larger max depth which can lead to overfitting. I replaced the minimum number of samples to split with the minimum number of samples in each leaf parameter. This would ensure the trees didn't base decisions off of small numbers of data points. I also decreased the max depth to 9 and increased the number of samples in each tree to 100% to remove bootstrapping since it was not necessary to decrease the training time.

Random Forest 4

- 90/10 train test split
- 50 trees
- Gini criterion
- Minimum number of samples in each leaf = 100
- Maximum number of samples in each tree = 100%
- Maximum number of features considered when making a split = 100%
- Maximum tree depth = 9

Random Forest 4 Analysis

This was a good middle ground between overfitting and underfitting. The training and testing errors were slightly off, but I was still getting the best test error for oversampled mean, and close to the best error for non-oversampled mean. I chose this as my final model.

Results

Method	Logistic Regression	Random Forest 1	Random Forest 2	Random Forest 3	Random Forest 4
Training Error (Mean)	0.579	0.667	0.678	0.453	0.6155
Training Error (Median)	0.0	1.0	1.0	0.0	0.0
Test Error (Mean)	1.515	0.593	0.581	0.557	0.597
Test Error (Median)	2.0	0.0	0.0	0.0	0.0
Oversampled Training Error (Mean)	0.539	0.668	0.648	0.359	0.548
Oversampled Training Error (Median)	0.0	1.0	0.0	0.0	0.0

Oversampled Test Error (Mean)	0.997	0.729	0.721	0.743	0.672
Oversampled Test Error (Median)	1.0	1.0	1.0	1.0	1.0

Conclusion

Overall I think I found models that will generalize well on new datasets and give predictions with good error rates. I decided to use a neural network for the first dataset and a random forest classifier for the second dataset.