A NumPy Model Zoo

Garrett Gibo
University of California, San Diego
ggibo@ucsd.edu

ABSTRACT

Using models from sklearn or any equivalent library has become increasingly easy; however, this ease of use can lead to lack of deep understanding. To alleviate this I have started the creation of a numpy modelzoo that implements some of the fundamental models that sklearn has to offer using only numpy. Linear regression, naive bayes, decision trees, and random forests are the models that were chosen because of the variety of knowledge that would be required for their implementation. The numpy modelzoo has been implemented in way that is meant to mimic the format of sklearn models. This report will cover the design, implementation, and results of my custom modelzoo.

CCS CONCEPTS

• **Computing methodologies** → *Supervised learning by classification*; *Supervised learning by regression*; Online learning settings; **Classification and regression trees**.

KEYWORDS

datasets, validation, NumPy, sklearn, pandas, regression, classification, trees, boosting

ACM Reference Format:
Garrett Gibo. 2020. A NumPy Model Zoo. In *Proceedings of* . ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

My model selection choice was driven by the desire to have fundamentals and also diversity. Linear regression is often one of the first topics covered, because it is such a fundamental concept to cover. Because I chose linear regression, I decided not to implement logistic regression as well. Moving forward I chose to implement naive bayes, because it is a good point to start before moving onto expectation maximization in the future. My final models were a decision tree and random forest classifier. These go hand in hand for classifications tasks, and I chose it because I wanted to implement a much more classification heavy model.

2 LINEAR REGRESSION

2.1 Design

As mentioned earlier, I all models to have the same functional workflow as models from sklearn. The main components from my linear regressor to use it are the fit, predict, and score methods. They mimic the behavior of sklearn almost exactly, making comparisons between them easy to make in the future. I implemented linear regression using stochastic gradient descent with a simple square error loss function.

The overall implementation of gradient descent was fairly standard. I am updating the weight (coefficients) every iteration, thus making the process stochastic. I though about doing some kind of mini-batch learning version of gradient descent, but ultimately decided against it primarily for computational restrictions. I implemented a very naive adaptive learning rate where the rate increases by a factor of 1.3 when the calculated loss decreased and decreased the learning rate by a factor of 0.9 when the loss increased. I didn't adjust these learning rate ratios for any further tuning, because it most likely would not have resulted in significant computational improvements.

Overfitting to data always needs to be thought about when designing and type of model, so for linear regression, I placed a very mild barrier that will simply early stop the gradient descent iterations when error increases. It is designed to stop after a single iteration of increased error; however, it could actually be more useful to allow some iterations to increase in error. If the loss increased for a some specified number of iterations, then that is when we should actually stop the iteration process. Once again the main reason I decided against this was for computational reasons.

I tested my model against a dataset geneerated from the formula $Y = 30 \times X + 20$ as well as a wine dataset from UCI [2]

2.2 Comparison

In order to test my model I benchmarked it against the default settings from the LinearRegression model in sklearn. For the custom generated dataset from an equation, the sklearn model performed almost perfectly in terms of speed and accuracy. The results from the sklearn model and my linear regrsesion model can be found in Table 1.My model on the other hand has some very unfortunate downsides. Because I implemented it using gradient descent, it is extremely slow when compared to its sklearn counterpart. In order to get accurate coefficients, I had to run the regressor for a full ten million iterations. I was originally hoping that one million iterations would be sufficient, but the results from Table 1 clearly show that the model performed much better with more iterations.

In addition to the equation dataset, I utilized the UCi wine dataset to benchmark my model. From the results in Table 2 we can see that although error between my model and the sklearn model were fairly close, the actual coefficients found were drastically different.

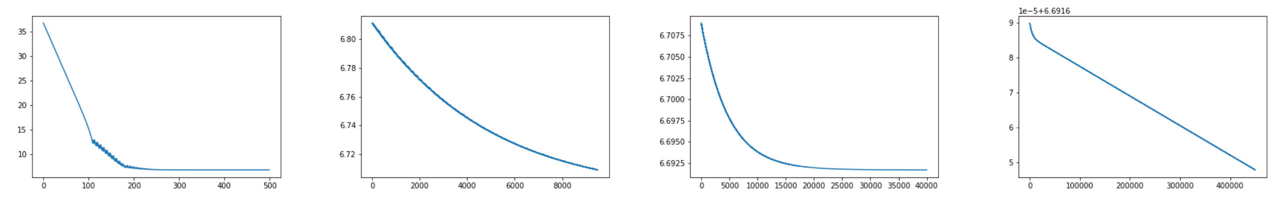

Figure 1: Loss over 500000 iterations

Table 1: Linear regresssion performance on equation

Model	Error	Coefficients
Custom-1Mil	-648.386	6.982, 30.019
Custom-10Mil	-12.532	19.752, 30.000
Sklearn	0	19.999, 30.0

I am not sure what the exact reason for this difference could be; one of my ideas regarding it was that my coefficients are being offset for some reason. I think this could be plausible because the sklearn intercept is -33.1523, while mine is 1.123.

Table 2: Linear regression on red wine dataset

Model	Error	Coefficients
Custom	805.648	1.123, 0.751, 0.361
sklearn	800.668	-33.152 34.822, 0.391

2.3 Interesting Findings

In Figure 1 we can see how loss changed over 500000 iterations of gradient descent when training on the UCI wine data. To actually plot the loss, I had to split the data to multiple different segments. In order to get a scope of the loss for the entire training process I had to split the iterations into four segments as seen in Figure 1. I found it very fascinating that there was such a significant difference in loss throughout the whole process.

3 NAIVE BAYES

3.1 Design

Similar to the linear regression model, I implemented Naive Bayes in a way that very closely mimics the format from sklearn. Naive Bayes at its simplest is just a counting problem, so the overall implementation is just determining what the likelihoods are for current($P(X_j|Y)$) and prior ($P(Y)$) labels. A key item to remember is that after counting the labels, they must be normalized to create an actual distribution. Once we have the two distributions, making a prediction is as simple as just iterating through the values and determining which combo has the highest probability.

3.2 Comparison

For the naive bayes classifier I used the balance scale dataset from the UCI machine learning repository [1]. The custom model did fairly comparable to the sklearn model in terms of both runtime and accuracy as seen in Table 3.

Table 3: Custom vs sklearn accuracy

Model	Accuracy	Runtime (s)
Custom	0.921	0.140
Sklearn	0.937	0.011

3.3 Interesting Findings

I didn't have any significant findings when implementing this model with just numpy. If anything, my main takeaway was how simple it was to create the model and achieve decent results (if matching sklearn is considered decent). Implementing this model has made it very clear exactly why it would perform extremely well if it was provided with infinite data. However, in reality, the lack of all scenarios is a critical weakness of the naive bayes model that results in cases where it would actually be impossible to classify some items if it had not seen it before.

4 DECISION TREE

4.1 Design

In terms of usage, the the decision tree has been implemented to have the same format as its sklearn counterpart. The tree is fundamentally based off the idea of choosing splits that maximize information gain. Information gain is defined as:

$$IG(Z) = H(X) - H(X|Z)$$

where, Z is the theshold to test at and H() is calculating the entropy. Using this formula we can calculate the information gain and how it changes as the threshold changes for a set of nodes.

Deciding splitting rules is also a critical component of a decision tree. If domain knowledge is provided, then I believe that splitting based off that knowledge would have been far more useful, but in my case I have simply made the tree split at the midpoint of all unique features. This allows the tree to be more generalized at the cost of runtime performance. The tree could potentially overfit to the training data if enough depth is allowed.

4.2 Comparison

The custom tree did very comparable to its sklearn counterpart. The results of its performance on the UCI wine dataset are showed in Table 4

4.3 Interesting Findings

The most interesting finding was that train error was always 0. Intuitively this could make sense that given enough time, a tree

Table 4: Custom Decision Tree vs Sklearn Accuracy

Model	Accuracy	Runtime(s)
Custom	0.628	5.620
Sklearn	0.665	0.026

will uniquely classify training label with a unique pure node. Since pure nodes by definition only have a single label, predictions for that vector in the future should always return the correct label. Although the training data did have impeccable accuracy, the risk of overfitting to that data is still a concern.

On the topic of overfitting, I found that the seed in which I split the data had a significant factor in the accuracy of the results. I initially created train test splits with a random seed of 0, and had the accuracy shown in Table 4. However, when I changed the seed to be 42, there was a drastic decrease in accuracy, shown in Table 5. Instead of doing a simple train test split, a k-fold cross validation system would probably show more representative results.

Table 5: Custom Decision Tree vs Sklearn Accuracy Seed = 42

Model	Accuracy
Custom	0.556
Sklearn	0.566

Unlike the Naive Bayes model, since a decision tree is based off thresholds instead of exact values, it is far better at handling new unseen data. Its classification method very much reminds me of how a k-nearest neighbors classifier works. Similar to the naive bayes model, if the decision tree is provided with infinite data, then it would literally be able to classify everything by using the rules it generates.

5 RANDOM FOREST

5.1 Design

I implemented a random forest to work in the simplest fashion by simply creating many decision trees and clustering their results to determine a prediction. In this naive form, the creating a random forest was as simple as creating a new random decision tree, storing this tree, and repeating for a specified amount of iterations. The main downsides that I saw of this method is memory usage and runtime.

Because we are literally just creating multiple decision trees, each one of those trees needs to be stored in memory. If each tree is also deep, then there is cause for concern regarding memory usage for a single random forest. I avoided this issue by simply limiting the number of trees to generate.

5.2 Comparison

The results of the custom model against the sklearn counterpart are shown in Table ??. The custom model actually had very slightly higher accuracy than the sklearn version; however, it ran significantly slower.

Table 6: Custom Random Forest vs Sklearn Accuracy

Model	Accuracy	Runtime
Custom	0.728125	243.928
Sklearn	0.721875	0.295

5.3 Interesting Findings

I was very surprised that my version of the random forest did better than the sklearn version. Creating a random forest inherently solves the problem that I addressed about decision trees and random initialization. Since we are creating many trees that are initialized with different seeds, the classification powers of the collective trees are amplified. Although my custom model did have very slightly better accuracy, it came with a significant runtime cost.

6 TAKEAWAYS

My main takeaway from completing all of these models is that sklearn is far more optimized than the most simple algorithms that I used to implement these models. Although I may have been able to get comparable results, the implementation that I used was often orders of magnitudes slower than their sklearn counterparts.

Overall I believe that I have learned a lot more about how each of these models fundamentally works. While the fundamentals do provide a baseline of performance, it is very clear that there is still plenty of optimization in the algorithms and their design to improve both accuracy and speed.kk

7 CITATIONS AND BIBLIOGRAPHIES

REFERENCES

[1] [n.d.]. UCI Machine Learning Repository Balance Data. ([n. d.]). https://archive.ics.uci.edu/ml/machine-learning-databases/balance-scale/balance-scale.data
[2] [n.d.]. UCI Machine Learning Repository Wine Data. ([n. d.]). https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv

ACKNOWLEDGMENTS

I'd like to thank Professor Shang for a very insightful quarter. Despite the circumstances, I believe that he was well adapted and taught the material very welll.