Spam Classification: Comparative Analysis of FFNN, Logistic Regression, and Random Forests

Evan Matel, Martin Jakhelln

December 2023

Abstract

In this report, we cover several machine learning methods, with the main focus being a Feed Forward Neural Network (FFNN) and Logistic Regression (LG), but also Random Forests. Our study involves applying these methods to the Spambase dataset from University of California-Irnvine (UCI) to correctly classify whether an email is spam or not. We found that for our logistic regression, a stochastic gradient descent using RMSprop gave us the best results and that in general, the FFNN gave us the best result with an accuracy of 0.925.

1 Introduction

Machine learning has blown up over the last decade, many people would not have known what AI was ten years ago, and it would have been quite difficult for someone to give you an example. Well, if you were to go back 15 years ago, and ask a person in the field of AI, which was at the time, quite insignificant, they might have given you a spam bot as an example. We wish to go deeper and understand some of the different ways to create such a spam bot and to do so, we will analyze the Spambase dataset taken from the University of Irvine (UCI). The dataset consists of roughly 4500 emails, the features include the frequency of certain words appearing, and the labels are binary, meaning either spam or not spam. We will explain several machine learning methods, including a Feed Forward Neural Network(FFNN), Logistic Regression, and Random Forest. We will then train models using each of these methods to accurately classify the data, hopefully within 95% of accuracy. We will then compare these models to discover their strengths and weaknesses. We will also compare our models to SKlearn's prewritten models. Sklear is a python library that is widely used in machine learning [5].

2 Method

2.1 Logistic Regression

Logistic regression is commonly used for binary classification problems, which as stated before, our data set is. The Sigmoid function(equation 1), also known as the logistic function is the key component of Logistic regression, this function takes an input number and scales it between 0 and 1, making it suitable for binary classification. However, we do not obtain the desired analytical answer from our logistic regression and we therefore have to deploy numerical approximations to get a desired answer. To implement these concepts, we define the sigmoid function and use that to create a logistic loss function(equation 2) and a cross-entropy loss function(equation 3). This will allow us to iteratively optimize our model and thereafter assess the accuracy of it. We included the cross-entropy loss function due to its ability to effectively model the connection between actual binary labels and probabilities, coupled with its strong compatibility with gradient descent optimization.

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{1}$$

$$\nabla J(\theta) = \frac{1}{m} X^T(\sigma(X\theta) - y) \tag{2}$$

$$J(\theta) = -\frac{1}{m}\left[y \log(\max(\sigma(X\theta), \varepsilon)) + (1-y)\log(\max(1-\sigma(X\theta), \varepsilon))\right] \tag{3}$$

Our optimization method will be an important part of making an accurate classifier. We are only going to use gradient descent(GD) and stochastic gradient descent (SGD) in this project, but we are going to use several variations of these optimization methods, hopefully figuring out which of the methods works best for this set. Starting a gradient descent is done by initializing theta with a random point within the space defined by our cost function. Further, we calculate the gradient, which indicates how theta's(θ) features will change. With this we navigate to the region that maximally reduces cost, how fast we move through this region is dependent on eta(η) also known as learning rate, and we have also included the regularization parameter lambda(λ) which changes our gradient by scaling it with the previous weights. These two parameters, lambda, and eta, will be the focus of our grid search for the optimal parameters, that hopefully will allow us to find the best possible solution. After this our algorithm will run until it finds a local minimum or until we have iterated n epochs.

SGD distinguishes itself from gradient descent by choosing small batches (mini baches) of the set, randomly selected, with replacement, hence the name stochastic, then iterating over those batches. This method aims to improve the efficiency of our descent in the defined space but also is one possible method for solving the problem of avoiding local minima, thanks to the randomness introduced.

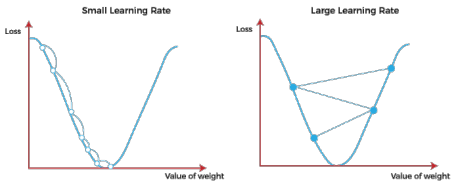

Figure 1: This Shows how a large or small learningrate affects how a gradient descent navigates through a 2D space [3]

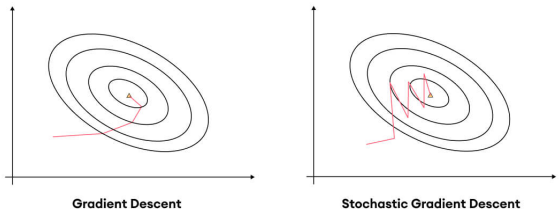

Figure 2: A visual representation of how the two methods differ from each other [4].

2.1.1 Adagrad

Adagrad, short for Adaptive Gradient Algorithm is an optimization method for both GD and SGD. This approach dynamically adjusts the learning rate for individual features based on the history of gradients. By doing this the feature with the steepest gradient receives smaller updates, while the shallower gradients receive bigger updates. Therefore, this method should improve the stability of our descent in the defined space, but it comes at a cost. Over time, the squared gradients can lead to diminishing learning, thus we can encounter convergence problems, namely, it never converges. It is worth mentioning that this is where we used the cross-entropy loss function.

2.1.2 RMSprop

RMSprop, or Root Mean Square Propagation, is an optimization algorithm designed to address challenges in gradient descent optimization. This method differs from the standard approach by adapting the learning rate during the training process. This is done by introducing the 'giter' parameter which consists of a decaying average of past squared gradients. By including this parameter we hope that this improves convergence and efficiency. The update rule for RMSprop is given as equation 4.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \odot g_t \tag{4}$$

where: $E[g^2]_t$ is the decaying average of squared gradients, g_t is the current gradient ϵ is a small constant to prevent division by zero.

2.1.3 Adam

Adam, or Adaptive Moment Estimation, includes two main methods that differ from a normal GD algorithm, the exponentially decaying average of past gradients gradients(first moment) and the exponentially decaying average of past squared gradients(second moments). What controls the decay rate are the two beta terms, where a higher value makes past gradients more relevant. As we have seen from previous methods, this method also aims to better tune the learning rate, and Adam has proven to be a versatile and all-around great optimization method[2].

$$\begin{gathered}
\mathrm{m}_t = \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot \nabla J(\theta_t) \\
\mathrm{v}_t = \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot (\nabla J(\theta_t))^2 \\
\hat{m}_t = \frac{m_t}{1-\beta_1^t} \quad , \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \\
\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon} \cdot \hat{m}_t
\end{gathered}$$

where: m_t and v_t are the first and second moments of the gradients $\hat{m}_t$ and $\hat{v}_t$ are the bias-corrected first and second moments ϵ is a small constant to prevent division by zero

2.2 Random forests

Random forest is a very robust method in machine learning that uses decision trees, a very simple and old method. At every point where the tree splits, which is called a decision point, only a random subset of features is chosen. The method is known for being very good at avoiding overfitting, a common problem in ML, meaning that it generalizes very well. The amount of decision trees that are made during training is commonly referred to as the "number of estimators." In addition to choosing a random subset of features, the algorithm also uses bootstrapping, which is where it chooses a random subset of the data with replacement and shuffles it, increasing the randomness, generalizing more, and therefore further avoiding overfitting. Before making a final decision, each tree "makes" its own decision and the final classification is made through a sort of voting process (see Figure 3).

2.3 FFNN

Neural networks are widely used algorithms in modern AI that are inspired by the brain's functioning. They are comprised of weights and biases which, under training, are adjusted to give wanted results. That is, the act of changing the weights and biases in an educated way is the training of the algorithm. Considering an NN with one hidden layer, we initialize it with random weights and biases, in the forward pass, inputs are multiplied by weights, biases are

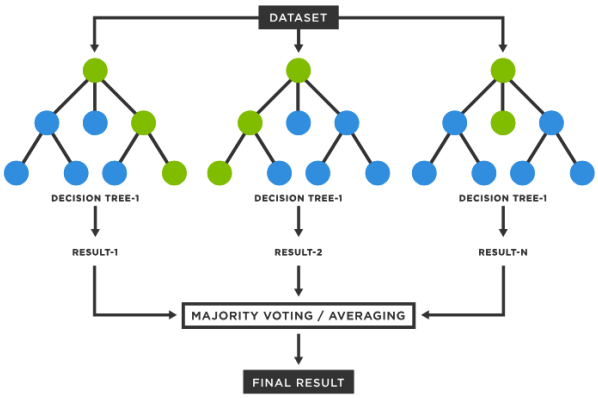

Figure 3: A visual interpretation for random forest. [6]

added, and then that sum is put inside an activation function (which is then sent to the next layer as the input in the case of several layers).

$$z = input \cdot W + B, \quad a = act_f unc(z) \tag{5}$$

Once the weights and biases are updated after the forward pass, we then start backpropagation, which adjusts the weights and biases based on the differences between predicted and true labels and moves backward through the network using gradient descent. Traditionally, backpropagation is done using the entire data set, but this can be computationally demanding when the dataset is very large so it's common to do what is known as batch training.

3 Results

3.1 Logistic Regression

Gradient Descent (GD)

Adagrad

Parameter	**Value**
Scaling	Standard
Best η	0.01
Best λ	0.1
Average Over	10 runs
Best Accuracy	88.6%

Table 1: Results from adagrad.

RMSprop

Parameter	**Value**
Scaling	Standard
Best η	0.01
Best λ	0.1
Average Over	10 runs
Best Accuracy	90.04%

Table 2: Results from GD RMSprop.

Adam

Parameter	**Value**
Scaling	Standard
Best η	0.01
Best λ	0.1
Average Over	10 runs
Best Accuracy	90.16%

Table 3: Results from GD Adam.

Stochastic Gradient Descent (SGD)
Adagrad

Parameter	**Value**
Scaling	Standard
Best η	0.001
Best λ	0.0001
$minibatch\_size$	128
Average Over	2 runs
Best Accuracy	90.665%

Table 4: Results from SGD Adagrad.

RMSprop

Parameter	**Value**
Scaling	Standard
Best η	0.001
Best λ	0.0001
$minibatch\_size$	128
Average Over	2 runs
Best Accuracy	90.77%

Table 5: Results from SGD RMSprop.

Adam

Parameter	**Value**
Scaling	Standard
Best η	0.001
Best λ	0.001
$minibatch\_size$	128
Average Over	2 runs
Best Accuracy	90.5%

Table 6: Results from SGD ADAM.

The provided results showcase the optimal parameters and the average accuracy achieved by running our program multiple times. All our models yielded satisfactory outcomes, with certain optimizers demonstrating superior performance. While we initially considered testing the best number of epochs, we opted against it, a decision we will elaborate on later. Additionally, due to similar considerations, we were unable to calculate the average accuracy for our Stochastic Gradient Descent (SGD) model over 10 runs.

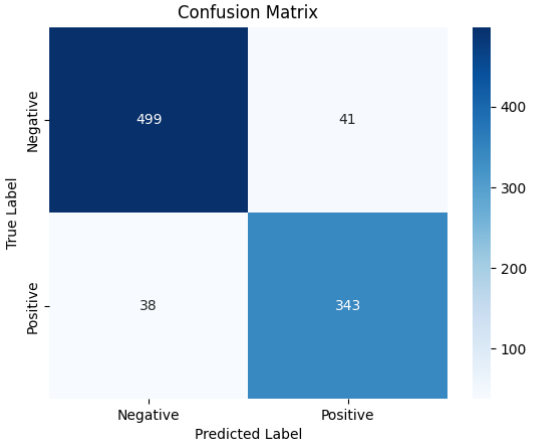

Figure 4: This plot shows the best run with the best optimizer for a gradient descent

In Figure 4, the confusion matrix plot illustrates the outcomes of the optimal run, providing valuable insights into the model's performance. This particular confusion matrix represents a run of GD utilizing the Adam optimizer. On the matrix's diagonal, we identify true negatives and true positives, while entries labeled as "positive negative" or "negative positive" indicate instances where the model incorrectly predicted positivity when the true label was negative, or vice versa.

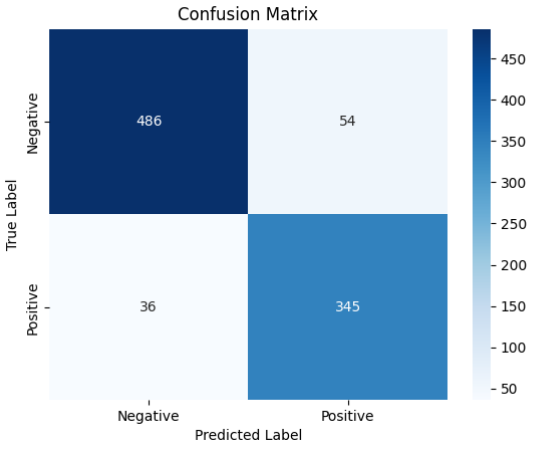

Figure 5: This plot shows the best run with the best optimizer for a stochastic gradient descent

In Figure 5, we present the confusion matrix plot depicting the outcomes of the optimal run using SGD with the RMSprop optimizer. Not surprisingly, the results indicate a slightly lower performance compared to our best GD run.

3.2 Random forests

Training Set

Num. of Estimators	**MSE**	**R2**
50	0.00678455	0.97145846
100	0.00651483	0.97259315
150	0.00636478	0.97322438

Table 7: Results from random forest training set.

Testing Set

Num. of Estimators	**MSE**	**R2**
50	0.05121302	0.78885496
100	0.05048426	0.79185953
150	0.05023855	0.79287257

Table 8: Results from random forest test set.

Our Random Forests model did not achieve the desired performance levels despite our expectation of good results when utilizing scikit-learn. We will delve into a discussion on this matter later.

3.3 Feed Forward Neural Network

Parameter	**Value**
Scaling	Standard
Hidden Layer Architecture	[30, 15]
Best η	1.93×10^{-5}
Best λ	1.38×10^{-2}
Best Epoch	105
minibatch_size	200
Best Accuracy	92.5%
SKlearn's Accuracy with Same Architecture	93.7%

Table 9: The parameters found to give the best result for our FFNN model.

We first did a grid search to find the best η and λ values for the model. After those were found we ran through several possible epoch values and found

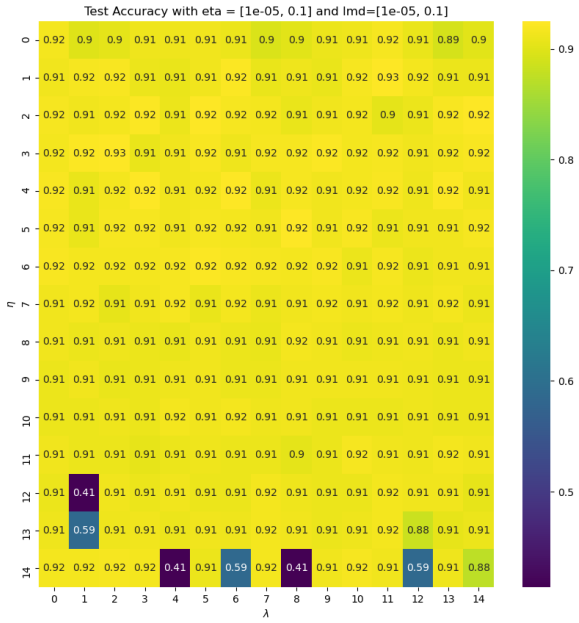

Figure 6: Shows the heat map at 200 epochs.

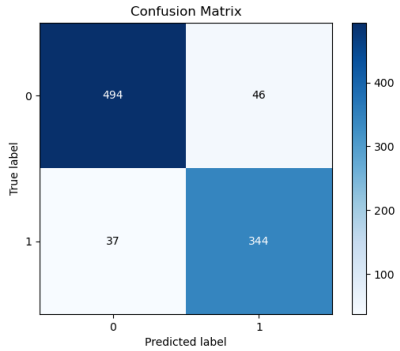

Figure 7: The confusion matrix for the FFNN model

no great difference between them. As one can see in Figure 6, the model is quite robust with the accuracy always around $90 - 92\%$. One can see the best parameters that were used in Table 9 and the accuracy of the model in Figure 7.

4 Discussion

4.1 Logistic Regression

As previously noted, we used two distinct loss functions in our analysis. When utilizing the logistic loss function, special attention was given to the thetas close to zero due to the undefined logarithm of zero. To address this, we introduced a small value, epsilon, to ensure numerical stability. Additionally, we incorporated the cross-entropy loss function to explore its potential performance benefits for our specific dataset. However, our findings indicated that both loss functions generally performed equally well. Consequently, we ended up mainly using the logistic loss function in our analysis.

As mentioned earlier, we opted not to conduct a search for the optimal number of epochs (n) due to runtime constraints. Although we performed a small sample test to identify a suitable value, the limitations on runtime prevented us from exploring the potential of better performance of different n epochs in combination with other parameters. This constraint also affected our ability to assess the average accuracy for our Stochastic Gradient Descent (SGD) model, as it would have extended the program's runtime beyond acceptable limits. While program efficiency improvements were possible, implementing them would have required a significant investment of time, which was not feasible in our case. One may wonder why we did or wanted to do averages over several runs. The reason lies in the fact that each time our model runs, it initializes with random weights, leading to different performances on each run. Therefore, incorporating averages over multiple runs becomes a crucial aspect of evaluating and understanding our model's overall performance.

We conducted scaling on our dataset and observed that standard scaling outperformed both non-scaling and min-max scaling methods. The improved performance of standard scaling can be attributed to the sensitivity of logistic regression to input features. Standard scaling effectively reduces the scale differences between features, which leads to easier convergence during training. It is also worth mentioning that if we did not scale the data, our optimizer, adagrad would not run due to convergence problems. We think this could be due Adagrad's method of incorporating past gradients, specifically, the cumulative sum of past gradients, which might lead to an excessive reduction in the learning rate over time, making convergence highly unlikely.

4.2 Random forests

After examining our results, we noticed that random forests didn't perform as well as expected. We suspect overfitting to be the cause since the training accuracy was much better than the testing accuracy, yet, this is quite interesting since random forest is known for avoiding overfitting due to the level of randomness involved. This could be due to having too many estimators. However, since random forests have many parameters to tune, extensive exploration of these parameters might yield improved results.

4.3 Feed Forward Neural Network

Our FFNN was quite successful, always landing around $90-92\%$ with barely any modification of hyperparameters. We spent some time trying to push past 93% with no avail, which when compared to SKlearn's model accuracy, at 93.7%, makes sense. One shouldn't expect to easily write a code better than SKlearn has done. Simultaneously, the lack of effect that the hyperparameters had on the model is very interesting. It's possible that the hyperparameters we didn't do a grid search on, like the batch size or the hidden layer architecture, were at a local maximum. It should also be mentioned that, when looking at the confusion matrices for all of our models, there were more false positives than false negatives, meaning the models incorrectly categorized emails as spam more than they incorrectly categorized them as not spam, which could be due to many reasons.

5 Conclusion

We have introduced and implemented several different machine-learning methods, which include Logistic Regression, Random Forest, and Feed Forward Neural Networks. These models were trained on the UCI Spambase data set. We found that our FFNN performed the best, with Logistic Regression being only slightly worse. Random Forest sadly underperformed, perhaps due to being poorly implemented. We almost reached our goal of 95%, missing the mark by less than 3%, but we still managed to make a great deal of insights from this endeavor.

References

[1] Hjorth-Jensen, Morten. "Applied Data Analysis and Machine Learning." Applied Data Analysis and Machine Learning - Applied Data Analysis and Machine Learning, compphysics.github.io/MachineLearning/doc/LectureNotes/build/html/intro.html. Accessed 17 Dec. 2023.

[2] Brownlee, Jason. "Gentle Introduction to the Adam Optimization Algorithm for Deep Learning." MachineLearningMastery.Com, 12 Jan. 2021, machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/.

[3] "Gradient Descent in Machine Learning - Javatpoint." Www.Javatpoint.Com, www.javatpoint.com/gradient-descent-in-machine-learning. Accessed 17 Dec. 2023.

[4] "Guide to Gradient Descent Algorithms." SuperAnnotate, www.superannotate.com/blog/guide-to-gradient-descent-algorithms. Accessed 17 Dec. 2023.

[5] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

[6] "Demystifying the Random Forest Algorithm for Accurate Predictions." Spotfire, www.spotfire.com/glossary/what-is-a-random-forest. Accessed 17 Dec. 2023.