Genre Classification Using Book Summaries

Zach Silver
Johns Hopkins University
3400 N. Charles St.
Baltimore, MA 21218
zsilver1@jhu.edu

Sindhuula Selvaraju
Johns Hopkins University
3400 N. Charles St.
Baltimore, MA 21218
sselvar4@jhu.edu

1 Introduction

The main aim of our project is to implement an algorithm to classify books into different genres based on their summary. Whenever we visit any bookstore, virtual or real, the books are arranged based on their genres to help the readers quickly find what they are looking for. In the Internet age it is no longer viable to manually classify the genres of huge numbers of new books every day. Therefore, we decided to use concepts that we learned in this course to implement a method to quickly classify books based upon their plot summaries into appropriate genres. Such techniques are often used by websites, not only classifying their inventory, but also to make suggestions to their users based on their previous searches. Genre classification applies not only to books but also to movies, music, and other forms of entertainment. Classification techniques similar to genre classification are also used to do things such as classify restaurants based on cuisine and courses based on their subject matter.

In order to accomplish our goal, we made use of an Artificial Neural Network, a very popular supervised learning method, to predict each book's genre (see the next section for a more in depth explanation of ANNs). ANN's are considered to be a method of supervised learning because they rely on the input of data that has already been collected and prepared by humans. In order to train this network, we fed it several thousand book summaries and their genres. We prepared each of these summaries for use by removing all punctuation and so-called stopwords, which are words that do not add any meaning to the text. We then broke each summary up into individual words, each of which was a feature for our network.

2 Neural Network Explanation

In order to correctly classify our data, we used a Feed Forward Artificial Neural Network with varying numbers of hidden layers and hidden nodes. Artificial Neural Networks, as their name implies, are modelled after the structure of the human brain. In the brain, there are billions of cells called neurons that are connected to each other via axons. These connections allow the neurons to transmit electrical signals in vast networks. Similarly, an Artificial Neural Network (ANN) consists of nodes, or neurons, organized in layers that are connected to each other.

Every Artificial Neural Network has at least two layers: an input layer and an output layer. The input layer contains a neuron for every feature of our data, where a feature is any information about the data that can be quantified in a single number. The output layer contains a neuron for each expected piece of output data. For example, if we are trying to predict housing prices using a neural network, some of the input features could be the number of rooms, the square footage, or the presence of a swimming pool, and the output would be the price of the house. Note that this example is a regression problem, where the output is continuous, but ANNs can also be used for classification, where each output is selected from a set of possible discrete outputs.

In addition to the input and output layers, there can also be any number of hidden layers in between.

We do not actively modify these layers, which is why they are "hidden," but they can add important functionality to our ANN, such as the ability to classify data that is not linearly separable. This means that ANN's with hidden layers are able to classify data that could not be separated by a simple line or plane. As the number and size of the hidden layers increase, the network gains greater flexibility, but also becomes more difficult to train and more prone to overfitting the training data. This means that the network can fit the training data too closely, so that it will not be useful for real world use.

Before explaining how ANN's work, we will first introduce some notation:

- Let $w_{ji}^{(l)}$ be the weight from node j in layer l to node i in layer $l+1$
- Let $w_{ji}^{(l)}$ be the weight from node j in layer l to node i in layer $l+1$
- Let $N_i^{(l)}$ be the net input of node i in layer l
- Let $O_i^{(l)}$ be the output of node i in layer l
- Let $\delta_i^{(l)}$ be the delta of node i in layer l
- Let L be the final layer of the network
- Let y_i be the true value of node i in the output layer

Each neuron in the neural network has some output value. For the neurons in the first layer, the output value is one of the features of our data. For the neurons in every other layer, the output value is calculated by first finding the net input. The net input is the dot product of the previous layers outputs and the corresponding weights (1). Then, this net input is fed into some activation function (we used the logistic function as our activation function, which is what we show in the examples below), and the output of the neuron is the output of this function (2). Therefore, each neurons output (except for those in the input layer) is dependent on all neurons in the previous layer. This process of finding all the neurons outputs is called forward propagation.

$$N_i^{(l)} = \sum_j O_j^{(l-1)} * w_{ji}^{(l-1)} \tag{1}$$

$$O_i^{(l)} = \frac{1}{1+\exp(-N_i^{(l)})} \tag{2}$$

Once we have used forward propagation to find the outputs of all neurons in the network, we must then update the networks parameters. This is done through a process known as back propagation. In back propagation, we find each neurons delta, and use this value to modify the weights and biases of the network. First, the deltas of the output layer neurons are computed:

$$\delta_i^{(L)} = y_i - O_i^{(L)} \tag{3}$$

Then, for every neuron in a hidden layer, the deltas are computed by summing over the deltas of the previous layer multiplied by the corresponding weights, the neuron's output, and the derivative of the activation function. (Note that the derivative of the logistic function is equal to $x(1-x)$, which is why we multiply by $(O_i^{(l)})(1-O_i^{(l)})$ below).

$$\delta_i^{(l)} = (\sum_j O_j^{(l+1)} * w_{ij}^{(l)})(O_i^{(l)})(1-O_i^{(l)}) \tag{4}$$

This process is somewhat similar to forward propagation, except it begins at the output layer instead of the input layer of the network (thus the name back propagation). Finally, every parameter of the network is updated based on the delta values of each neuron and the learning rate (LR).

$$w_{ji}'^{(l)} = w_{ji}^{(l)} + O_j^{(l)} \delta_i^{(l+1)} * LR \tag{5}$$

The math behind back propagation is somewhat complex. Essentially, we want to find the values of the network weights that decrease the total error of the network. This can be done by computing the partial derivatives of the cost function with respect to each weight in the network. Then, the chain rule is used to compute the partial derivatives of weights in the previous layer, etc. By doing this, back propagation ensures that our network will now make a more educated guess when predicting future values.

3 Project Implementation

3.1 Data Collection and Preparation

We collected our book summary data from the Carnegie Mellon University Book Summary Dataset, which contains plot summaries and meta-information for over 15,000 books.

The data that we downloaded from the above source contained a lot of information that we did not need. It included the Wikipedia ID, the Freebase ID, the title, the author, the publication date, the genres, and the plot summary. In the first data preparation step we wrote a python script that removed all metadata except for the book title, genres, and the plot summary. We wrote these to a new file, one per line, with each piece of data separated by "|" so that they could be easily separated. In this step we also utilized the Python library NLTK to lemmatize our summary. This means that we reduced all words to their base form, removing all derivational or grammatical affixes from the word. (We also considered taking synonyms into account, but decided this would be too complex and would not be worth it.)

The second preparation step removed stopwords from the book summaries. Stopwords are words that do not add any meaning to the text, and are usually pronouns, articles, and other grammatical words. In this step we also removed all genres except for the most relevant one, chosen from twelve possibilities. Our possible genres were: Fiction, Speculative Fiction, Science Fiction, Fantasy, Mystery, Suspense, Crime Fiction, Horror, Romance, Children's Literature, Historical Fiction, and Non-fiction. After doing this, each book belonged to only one of twelve genres, which we thought would make classification easier.

In our final preparation step, we calculated the input values for each word in each summary. Initially, we were going to input the number of times each word appeared in a summary as the input for that summary, but eventually we decided to take a slightly different approach. We used a method called *tf-idf* (term frequency–inverse document frequency). The term frequency of a word is the number of times a word appears in a given summary. The inverse document frequency is calculated by taking the total number of summaries divided by the number of summaries the word appears in appears in. The *tf-idf* is simply the product of the *tf* and the *idf*. We chose to use this method, because it gives gives lower weight to words that appear frequently in all summaries. This makes sense because these words are likely very common, and therefore probably are not going to help much in classification.

In this step we also removed all but the top 3,500 most common words in our data, so that our input vectors would not be too large (there were initally over 50,000 unique words in our data). As a result, the input layer of our neural network contained 3,500 nodes, each of which corresponded to the *tf-idf* value of a word.

3.2 Code Implementation

3.2.1 Language and Libraries Used

For our project, we used version 2.7.12 of the Python Programming Language. We also used the NLTK library, the NumPy library, and the SciPy library.

3.2.2 Program Design

In order to classify our data, we designed an Artificial Neural Network from scratch. Our neural network implementation consists of two classes, a class for the neurons, called `Node`, and a class for the network itself, called `NeuralNetwork`. The `NeuralNetwork` class takes several parameters, including the number of input, hidden, and output nodes, the number of hidden layers, and the learning rate of the network. When the `NeuralNetwork` class is first instantiated, the method `populateNetwork` is called. This method fills the instance variable `network` with lists representing each layer of the network, and then fills each of these layers with the correct number of `Node` objects. Now the network is ready for training. The networks we created for classifying our books each contained 3,500 input nodes and twelve output nodes, as well as zero, one, or two hidden layers.

In order to train a `NeuralNetwork` instance, we must first call the method `forwardPropagate` with the input vector as our parameter. This method performs forward propagation layer by layer by computing each node's output (each node has a method called

`getOutput` that computes and returns the dot product of the weight vector and the previous layer's outputs). Eventually, we reach the output layer, and the network returns each of the output nodes' outputs.

After this process is completed, we can then call the `backPropagate` method. This method takes the true label and the predicted label for a given input as parameters. This method first computes the delta values for the output nodes, and then proceeds to back propagate these values until it reaches the input layer. Once the delta values for all the nodes are calculated, the `updateNeuronParams` method is called, which loops through every node in the network and updates its parameters. (Note: each `Node` object contains a vector of weights, one weight for each node in the previous layer.)

These two methods are called repeatedly for each example in our training data. Then, once the network has been sufficiently trained, we can run the `forwardPropagate` method on any test example, and it should in theory output the correct genre.

3.3 Problems and Future Scope

The main problem with our project was the performance of our neural network implementation. When there were no hidden layers, the network performed reasonably well during both training and testing. This makes sense, because a neural network with no hidden layers is essentially a multi-class perceptron.

However, as soon as we introduced a single hidden layer, the performance of the network took a massive hit, especially when there was a large number of hidden nodes. When a second hidden layer was added, the performance took an even greater hit, sometimes taking hours to train. As a result, we do not have as many results from the network as we would like (this will be explained further in the results section). Therefore, one simple way our project could be improved could be to rewrite the neural network code in a different programming language. Python, though a very easy to use and useful language, is an interpreted language, which means that it performs many times slower than compiled languages such as C++ or Java. As a result, we believe that if our code was written in either of these languages, performance would not have been as much of an issue. Also, the performance of our network could have been improved if we had implemented it in a different programming paradigm. Our object oriented approach, while easy to implement, probably was not the most efficient, because often the creation of many thousands of objects can slow down a program.

Another albeit less serious problem was that the data initially contained far too many possible genres. In fact, before we prepared the data, there were over 100 possible genres for each summary, many of which were extremely rare. As a result, we had to assign many of the books to one of the twelve most common genres, often assigning them the "fiction" label if they did not fit any of the other categories. This resulted in a rather uneven distribution of genres, which could have negatively impacted our results. Ideally, each genre would have an equal number of examples, so that the network would not simply predict the most common genres.

If we were to continue this project in the future, there are a number of paths we could take to improve it. One of the most interesting of these would be experimenting with new ways to parse our input data. For example, we could attempt to use the NLTK library to find synonyms in the text, and then treat these synonyms as the same word. This approach would not be guaranteed to improve our results, because we might miss out on shades of meaning that could help in our classification. However, it would be interesting to see how using synonyms would affect our results and the performance of our network. Another interesting change would be if we included apostrophes in our data instead of removing them. The use of apostrophes and contractions often indicates a more casual piece of writing, so the inclusion of apostrophes could also have been useful in determining the genres of our data.

4 Results

Our results are presented below. Note that in each of the trials below, our neuron network contained 3,500 input nodes and 12 output nodes. Each input node corresponded to a feature (the *tf-idf* of each word in a summary), and each output node corresponds to a different genre. With a single hidden layer, the network had 55 hidden nodes, and with two hidden layers, the network had 27 nodes in each hidden layer.

We decided to split the data into two parts: training and testing. For almost all of the results below, we used 90% of our data for training and 10% of our data for testing. We found that this ratio of training to testing data gave us the most accurate results (see below for results with 60% training and 40% testing). We also randomly selected which instances would be used for each purpose. (Note that we have many more trials with the network without any hidden layers, because it was much faster to train.)

Table 1: Baseline Results

Prediction	Prediction Accuracy
Random Labels	7.11%
Most Common Label	11.32%
Random Top 3 Labels	15.25%

Table 2: Results with 0 Hidden Layers and 60% training

Learning Rate	Prediction Accuracy
0.5	34.03%
1.0	34.86%
2.0	35.32%
5.0	29.25%

Table 3: Results with 0 Hidden Layers

Learning Rate	Prediction Accuracy	Training Iterations
0.5	43.33%	1
1.0	46.09%	1
2.0	41.95%	1
5.0	32.84%	1
0.5	48.94%	2
1.0	48.57%	2
2.0	43.61%	2
5.0	32.23%	2

Table 4: Results with 1 Hidden Layer

Learning Rate	Prediction Accuracy
0.5	22.45%
1.0	29.07%
2.0	29.44%
5.0	28.33%

Table 5: Results with 2 Hidden Layers

Learning Rate	Prediction Accuracy
0.5	21.71%
1.0	27.51%
2.0	28.24%
5.0	28.61%

Though these results may not seem impressive at first, there are a number of things to consider before dismissing them. Firstly, we must remember that our results should be compared to the baseline results in table 1. Even the network's worst result outperformed all three baseline tests, which shows that the network in fact has not simply learned to output the most common genre every time.

In fact, when we looked at a single example output, we saw that the network may be even more effective than the above results show. With zero hidden layers, the first book in the dataset, *Animal Farm* by George Orwell, was classified incorrectly as Speculative Fiction, when it should have been classified as Children's Literature. In the original dataset, each book was classified into several genres, and we simply assigned each book to its "primary" genre. *Animal Farm* originally was classified as both Children's Literature and Speculative Fiction, which means that the network in some ways was correct. One thing we could have done to improve our accuracy would be to enable books to be assigned to multiple genres, as there are some books, such as *Animal Farm*, that do not easily fit into one genre or another.

It may also seem strange that the network with zero hidden layers performed nearly twice as well as with one or two hidden layers. In our opinion, the reason for this is simply that we did not have enough hidden nodes. Ideally, the number of hidden nodes should be somewhere around the mean of the number of input and output nodes. The reason that we have so few hidden nodes is that our network's performance was simply too slow to include any more. We tried to run the network with 1,750 hidden nodes in a single hidden layer, but even after 12 hours (we left it running over night), the network was not finished training. We continued experimenting and found that 55 hidden nodes in a single hidden layer was the most we could do in a reasonable amount of time. Similarly, we found that with two hidden layers, the maximum number of hidden nodes we could have in each layer is 27. We believe that so few hidden nodes contributed to the poor ac-

curacy of the deeper networks, because they made the networks more prone to underfitting. This is because underfitting tends to happen when there is not enough complexity in the model, and too few hidden nodes is an example of too little complexity.

Interestingly, as we increased the number of hidden layers in the network, the ideal learning rate also seemed to increase. In the case of zero hidden layers, the best learning rate was about 1.0, whereas with two hidden layers, we got our best results with a learning rate of 5.0. This makes sense, because larger neural networks tend to train more slowly, so an increased learning rate may have simply sped up the learning process.

Finally, we noticed that we got the best results with no hidden layers when we increased the training iterations from 1 to 2. (We could not increase the training iterations for the networks with hidden layers because they took too long to train.) This may be an expected result, but it also shows that the network not prone to overfitting as long as we keep the number of training iterations low. As a result, we knew that we did not need to implement an advanced regularization technique such as L2 regularization, because there was little danger of overfitting.

5 Comparison to Proposal

- We did complete everything in the "Must Achieve" section of our project proposal. We implemented a neural network from scratch in python using NumPy and SciPy. We implemented several neural networks with various parameters and we compared them.

- We did complete almost everything in the "Expected to Achieve" section of our project proposal. Implementing a neural network from scratch gave us a much deeper understanding of neural networks and how they work in practice. One thing we did not implement was L2 regularization. As it turns out, a much bigger problem for our neural network was underfitting, because we simply did not have the computation power to train a large neural network. Since L2 regularization is done to prevent overfitting and not underfitting, there was no reason for us to implement it in our network.

- We partially completed the contents of our "Would like to Achieve" section in our project proposal. While we did not implement unsupervised pre-training as described in class, we did use the *tf-idf* method to improve the expected results of our neural network before we trained it. In a way, this could be thought of as a way to pre-train our network. While we did modify our input data quite heavily from its original form, we did not really experiment with changing our input data and seeing how it affected our results. We could have done this if we had for example taken synonyms or punctuation into account, but we decided to keep our project simpler and focus on implementing our neural network correctly.

- One thing that we were not able to do was to effectively compare neural networks with various numbers of hidden layers. This is because the networks with one or more hidden layers took an extremely long time to train, sometimes close to three hours. As a result, it was unfortunately simply impractical for us to run a lot of experiments on these networks.

References

Charu C. Aggarwal and ChengXiang Zhai. 2012. "A Survey of Text Classification Algorithms." *Mining Text Data*, Springer, New York, NY, 2012.

Lodhi, Huma et al. "Text Classification Using String Kernels." *Journal of Machine Learning Research*, vol. 2, Feb. 2002.

Stamatatos, E. et al."Text Genre Classification Using Common Word Frequencies." *Proceedings of the 18th Conference on Computational Linguistics*, vol. 2, 31 July 2000, pp. 808814.

Goldberg, Yoav. "A Primer on Neural Network Models for Natural Language Processing." *Journal of Artificial Intelligence Research*, vol. 57, 16 Nov. 2016, pp. 345420.