CSYE 7245 BIG DATA SYSTEMS AND INTELLIGENCE ANALYTICS

AUTHOR BASED TEXT CLASSIFICATION USING LSTM

Names: Ashwanth Ramji, Prasanna Hari Kumar

Introduction

In this project we are trying to predict the Authors of 3 books based on the text written by them. This is a Kaggle Competition called Spooky Author Classification where the task is to predict Author based on text lines from the books. For this particular problem, we have decided to use LSTM (Long short-term Memory) which is a special Recurrent Neural Network. Why are we trying to use deep learning for Natural Language Processing (NLP)? Natural Language Processing is nothing but creating systems that process or understand language to perform certain tasks. Deep learning has seen incredible progress and its applications to NLP tasks have been one of the biggest areas of deep learning research.

Project Description

The project deals with identification of authors based on their style of writing using NLP, the project is an adaptation of the Kaggle competition – Spooky author classification. To implement the project we have utilized concepts such as Word-vectors, Recurrent Neural Networks and Long short-term memory units and implemented these concepts using Jupyter notebook and Tensorflow with GPU utilization.

There were other forms of implementation but we decided to go with the above method so as to learn and making a breakthrough with respect to the conventional forms.

About the Data and The Visualization

The data originally provided consisted of a csv file containing the test lines and author name for that text. We had to change the format from csv to doc format, so that we cab specify the text in a document and all those documents are provided under one folder having one author, so we could split the texts based on author from csv file and store in a folder having the author name as folder-name. We performed this operation so that the the process would make the training and testing easy.

After performing the above operation, we had to perform some pre-processing steps. To remove various forms of impurities from the dataset.

Once the data pre-processing is complete, we had visualized the data to understand any anomalies in the data.

More info on data pre-processing + images of visualization

Exploratory Data Analysis

	id	text	author
0	id26305	This process, however, afforded me no means of...	EAP
1	id17569	It never once occurred to me that the fumbling...	HPL
2	id11008	In his left hand was a gold snuff box, from wh...	EAP
3	id27763	How lovely is spring As we looked from Windsor...	MWS
4	id12958	Finding nothing else, not even gold, the Super...	HPL

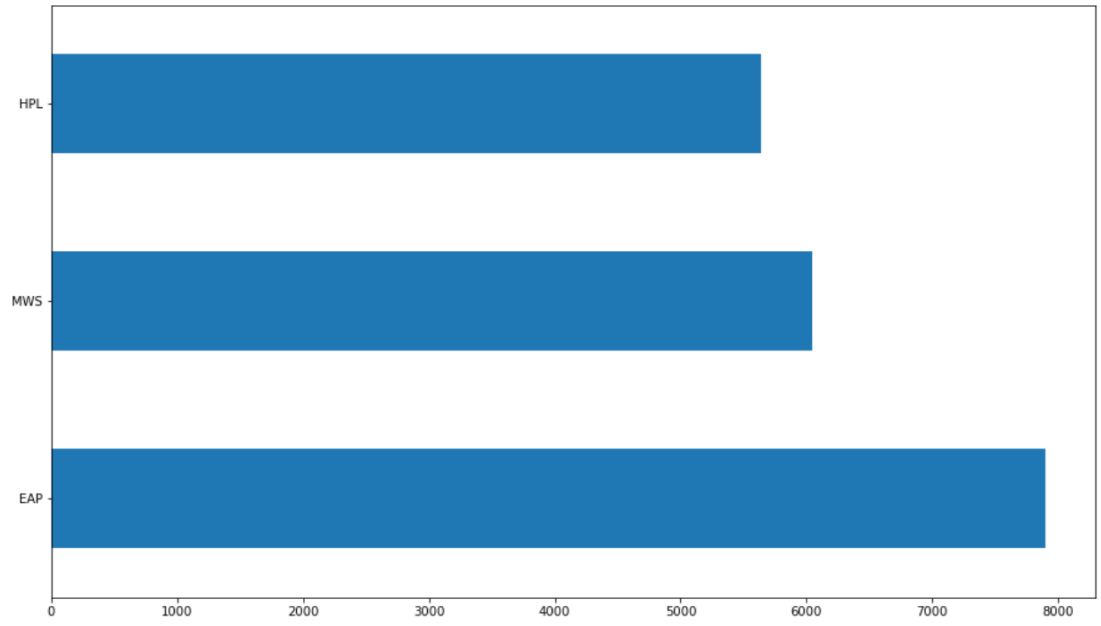
The three columns in our dataset are

1. id which is a unique identifier of each text line.

2. text line from the book.

3. author of the book.

When we check for null values, we find that there are no null values. To see the count of each Author in the dataset we visualize them using a bar chart



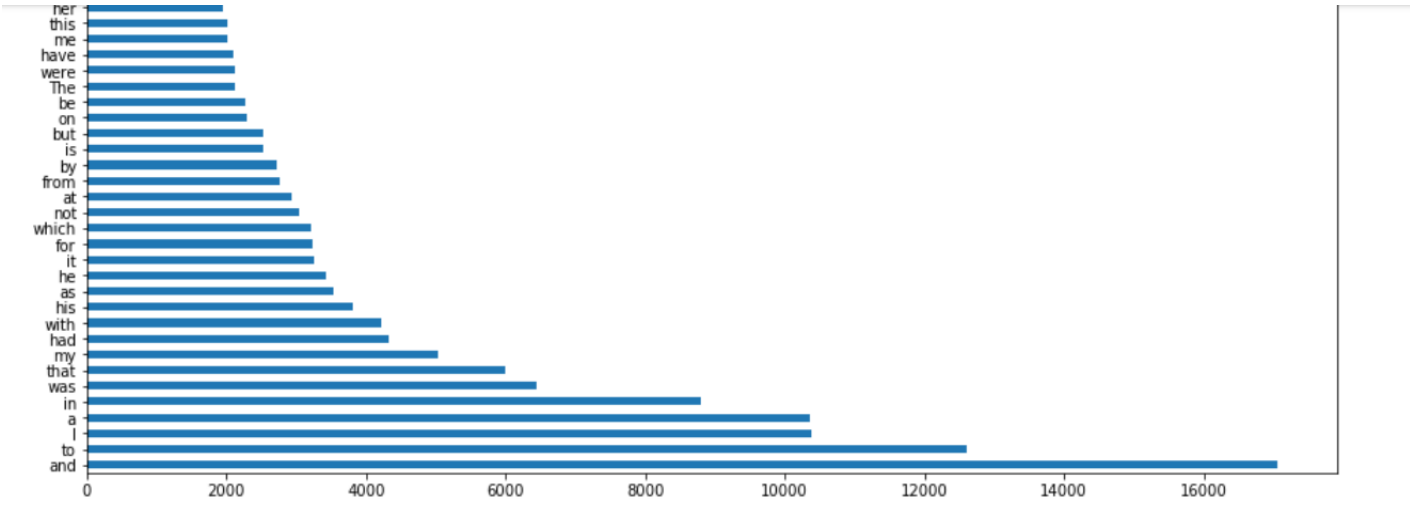
This shows the count of texts for each author.

1. EAP (Edgar Allen Poe) : 7900

2. HPL (HP Lovecraft) : 5635

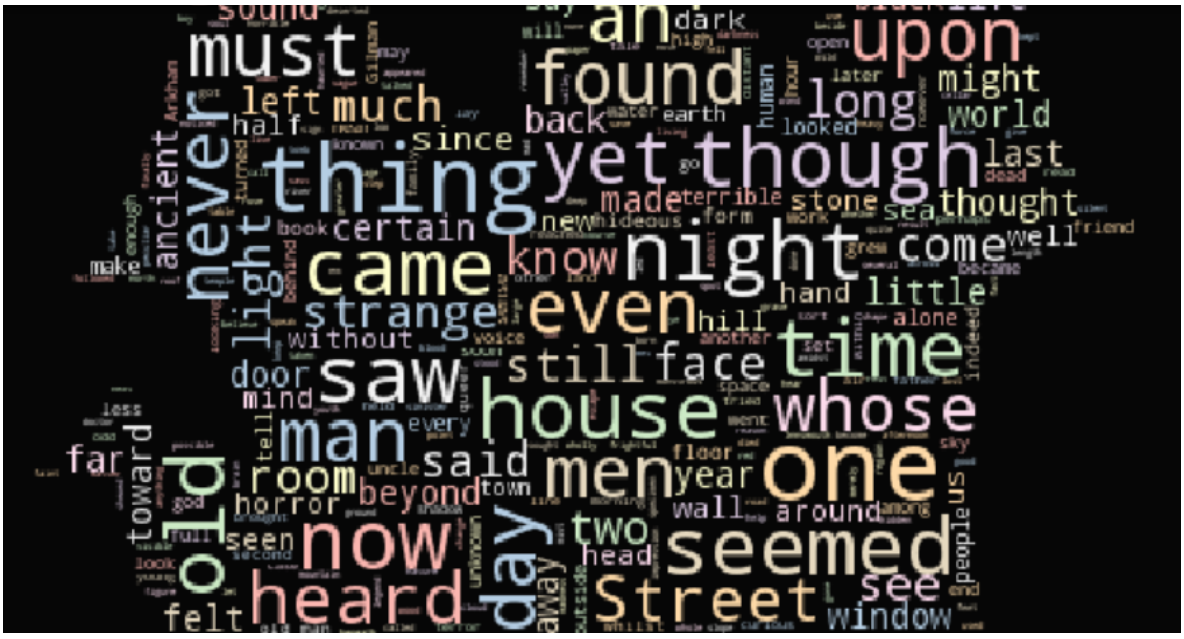
3. MWS (Mary Shelly) : 6044

Next we Visualize the top 50 words that occur frequently



These words are not only the common words that occur in the spooky author books but in almost all the books. So we could elminate these words and all common words so that our machine doesn't spend time learning them.

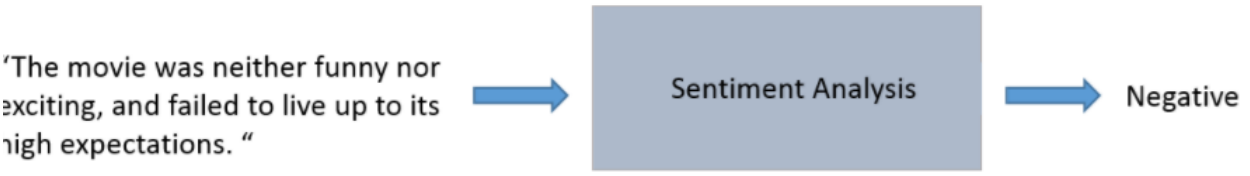
After putting the words into a list we plot the most occurring words in a word cloud



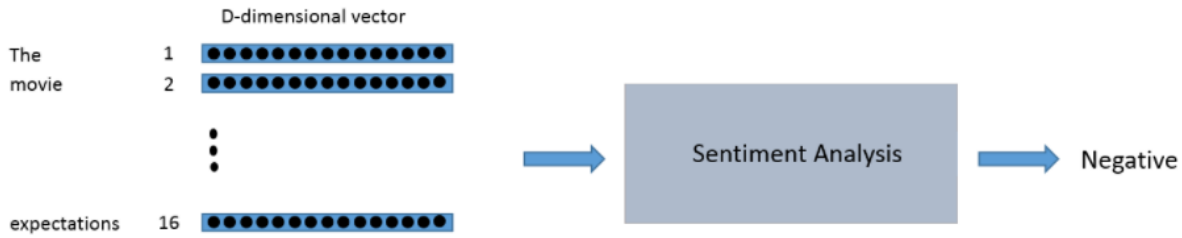
Now the data is ready to be processed further to generate the vectors.

Word Vectors

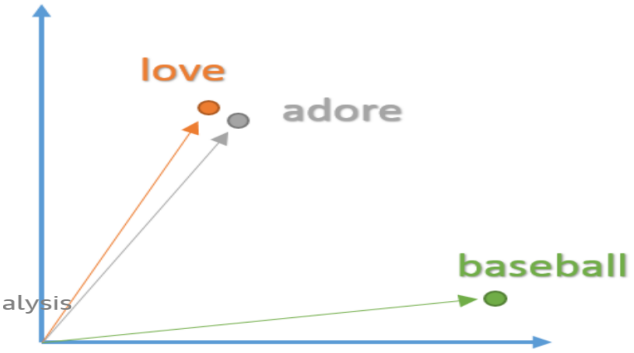
The common theme of inputs that have to be provided is scalar values or matrices of scalar values. When we think of NLP tasks a data pipeline like the following comes to mind.



We will need to convert a single string input to the form of vectors and cannot performs operations like dot product or back propagation on a single string. Thus we convert the input to a vector.



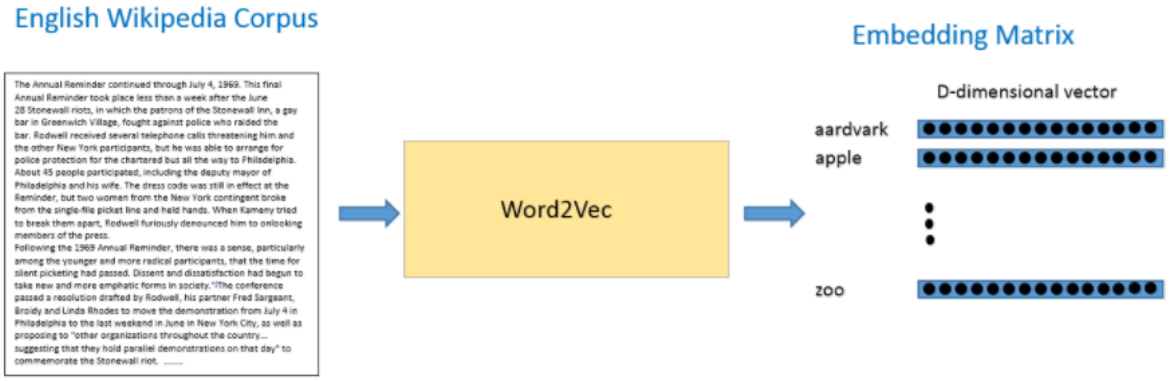
You can think of the input to the text classification modules as being 16 x D dimensional matrix. We have performed this step to represent the word and its context, meaning and semantics. For example love and adore reside in relatively the same area in the vector space since they both have similar definitions and both used in similar contexts. The vector representation of a word is also known as a word embedding.



GloVe Model

To create these word embeddings, we have used the GloVe model. In short GloVe is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space.

For further understanding, the model creates word vectors by looking at the context with which words appear in sentences. Words with similar contexts will be placed together in the vector space. Context is very important when considering the grammatical structure in sentences. Most sentences will follow traditional paradigms of having verbs follow nouns, adjectives precede nouns, and so on. For this reason, the model is more likely to position nouns in the same general area as other nouns. The model takes in a large dataset of sentences (English Wikipedia for example) and outputs vectors for each unique word in the corpus. The output of a Word2Vec model is called an embedding matrix.



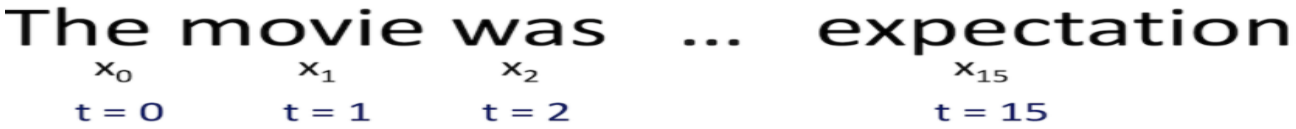
The embedding matrix will contain vectors for every distinct word in the training corpus, traditionally embedding matrices can contain over 3 million word vectors. The Glove model is trained by taking each sentence in the dataset, sliding a window of fixed size over it, and trying to predict the center word of the window, given the other words. Using the loss function and optimization procedure, the model generates vectors for each unique word. Thereby we can conclude that the inputs to any deep learning approach to an NLP task will likely have word vectors as input.

Recurrent Neural Networks(RNN)

Now that we have our word vectors as the input, let's look at the actual network model setup we are going to make. The Unique aspect of NLP data is that there is a temporal aspect to it. Each word in a sentence depends greatly on ehat came before and comes after it. In order to account for this dependency, we use the RNN

The recurrent neural network structure is a little different from the traditional feed forward NN that we are accustomed to seeing. The Feedforward network consists of input nodes, hidden units and output nodes.

The main difference between the feed forward neural networks and the recurrent ones are the temporal aspect of the latter. In RNN, each word in an input sequence will be associated with a specific time step. In effect, the number of the time steps will be equal to the max sequence length

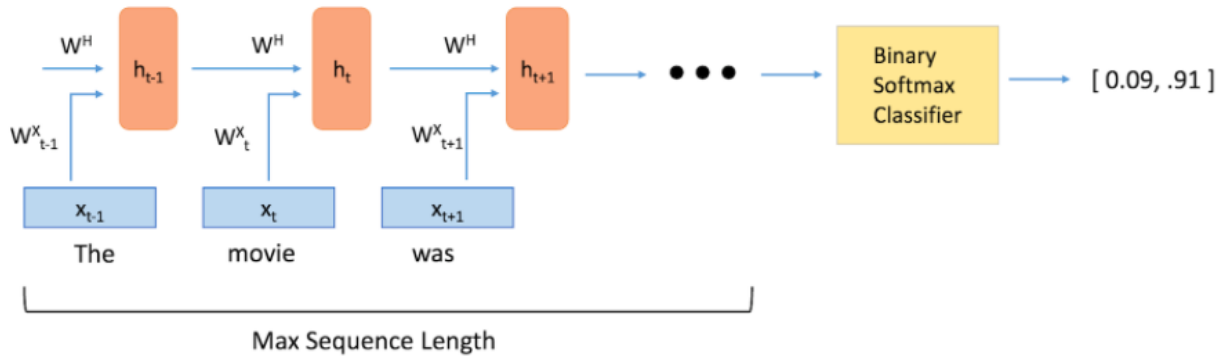


There is a new component that needs to be understood called the hidden state vector. That encapsulates and summarizes all the previous time steps ie., a vector that summarizes information form previous time steps. The hidden state is a function of both the current word and the hidden state vector at the precious step. They will be both be put through an activation function like sigmoid or tanh.

There is also a recurrent weight matrix which is multiplied with the hidden state vector at the previous steps and the weight matrix will stay the same across and the weight matrix is different for each input.

The weight matrices are updated through an optimization process called backpropagation through time.

Finally the hidden state vector at the final ste p is fed into a binary softmax classifier where it is multiplied by another weight matrix and put through a softmax function that outputs values between 0 and 1, effectively giving us the probabilities of the author.



LSTM – Long Short Term Memory Units

Long Short Term Memory Units are modules that you can place inside of recurrent neural networks. At a high level, they make sure that the hidden state vector h is able to encapsulate information about long term dependencies in the text. As we saw in the previous section, the formulation for h in traditional RNNs is relatively simple. This approach won't be able to effectively connect together information that is separated by more than a couple time steps. We can illustrate this idea of handling long term dependencies through an example in the field of question answering. The function of question answering models is to take an a passage of text, and answer a question about its content. Let's look at the following example.

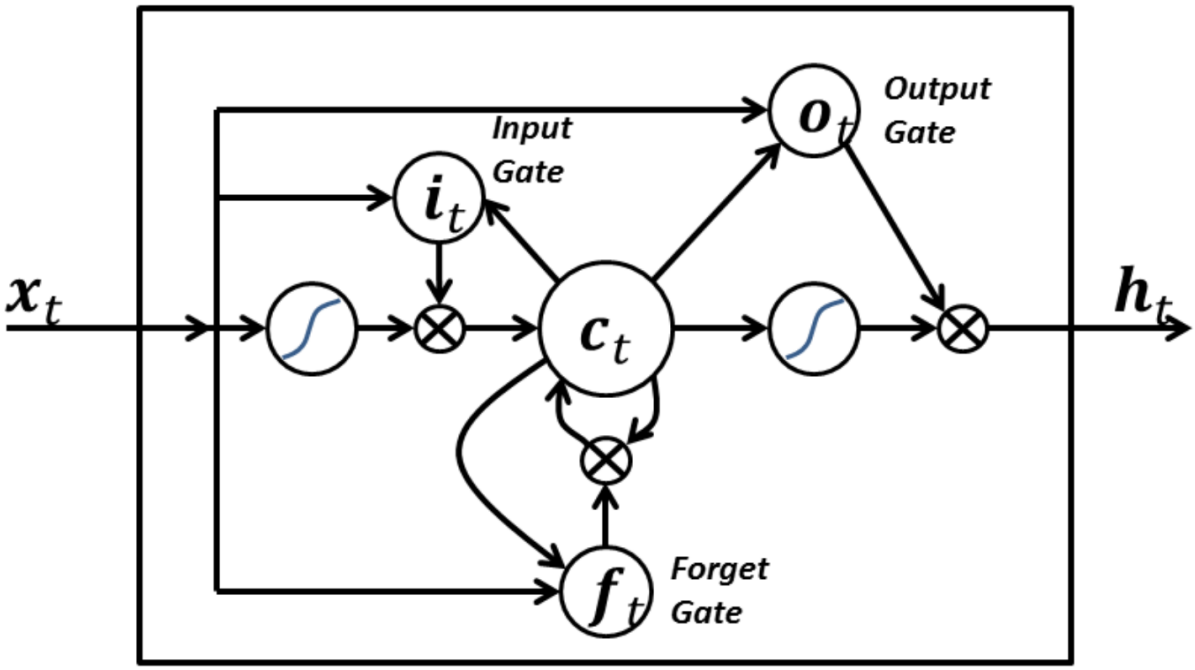
Passage: "The first number is 3. The dog ran in the backyard. The second number is 4."

Question: "What is the sum of the 2 numbers?"

Here, we see that the middle sentence had no impact on the question that was asked. However, there is a strong connection between the first and third sentences. With a classic RNN, the hidden

state vector at the end of the network might have stored more information about the dog sentence than about the first sentence about the number. Basically, the addition of LSTM units make it possible to determine the correct and useful information that needs to be stored in the hidden state vector.

Looking at LSTM units from a more technical viewpoint, the units take in the current word vector x_t and output the hidden state vector h_t. In these units, the formulation for h_t will be a bit more complex than that in a typical RNN. The computation is broken up into 4 components, an input gate, a forget gate, an output gate, and a new memory container.



Each gate will take in x_t and h_{t-1} (not shown in image) as inputs and will perform some computation on them to obtain intermediate states. Each intermediate state gets fed into different pipelines and eventually the information is aggregated to form h_t.

Each of these gates can be thought of as different modules within the LSTM that each have different functions. The input gate determines how much emphasis to put on each of the inputs, the forget gate determines the information that we'll throw away, and the output gate determines the final h_t based on the intermediate states.

Now completing the above example explanation – The model will likely not have an impact on the answer to the question, and thus the unit will be able to utilize its forget gate to discard the unnecessary information about the dog, and rather keep the information reading the numbers.

Now that we have the basics setup let's start the process of putting up all the pieces together.

Implementation Procedure

Loading Data

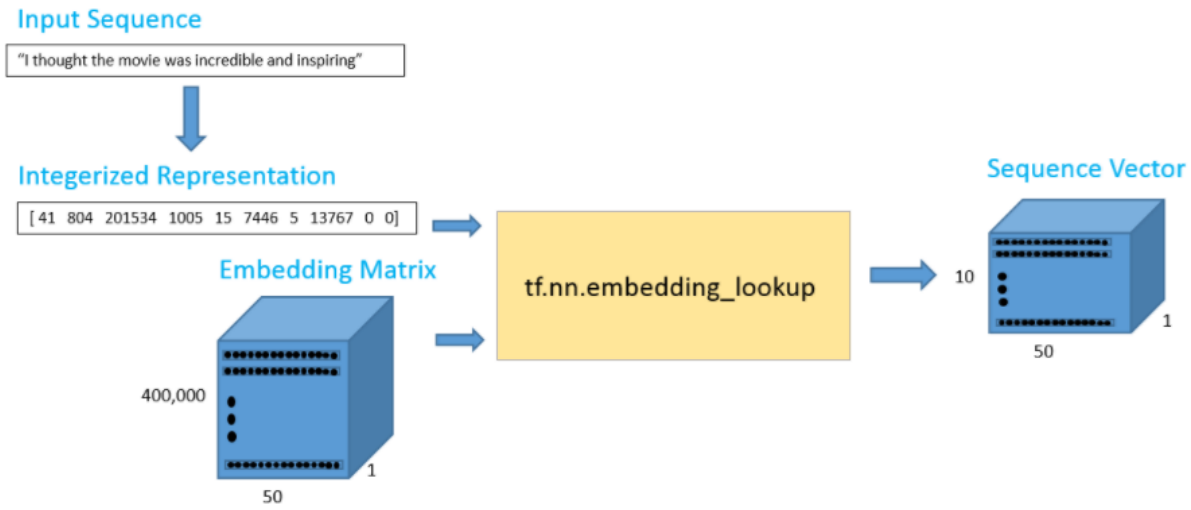
First, we want to create our word vectors. For simplicity, we're going to be using a pretrained model.

In an ideal scenario, we'd use word2vec model vectors, but since the word vectors matrix is quite large (3.6 GB!), we'll be using a much more manageable matrix that is trained using GloVe, a similar word vector generation model. The matrix will contain 400,000 word vectors, each with a dimensionality of 50.

We are going to be creating two different data structures one will be a python list with the 400,000 words, and one will be a 400,000 x 50 dimensional embedding matrix that holds all of the word vector values.

Now that we have our vectors, our first step is taking an input sentence and then constructing its vector representation. Let's say that we have the input sentence "I thought the movie was incredible and inspiring". In order to get the word vectors, we can use Tensorflow's embedding lookup function. This function takes in two arguments, one for the embedding matrix (the wordVectors matrix in our case), and one for the ids of each of the words. The ids vector can be thought of as the integerized representation of the training set. This is basically just the row index of each of the words.

The data pipeline can be illustrated below:



Before creating the ids matrix for the whole training set, let's first take some time to visualize the type of data that we have. This will help us determine the best value for setting our maximum sequence length. In the previous example, we used a max length of 10, but this value is largely dependent on the inputs you have.

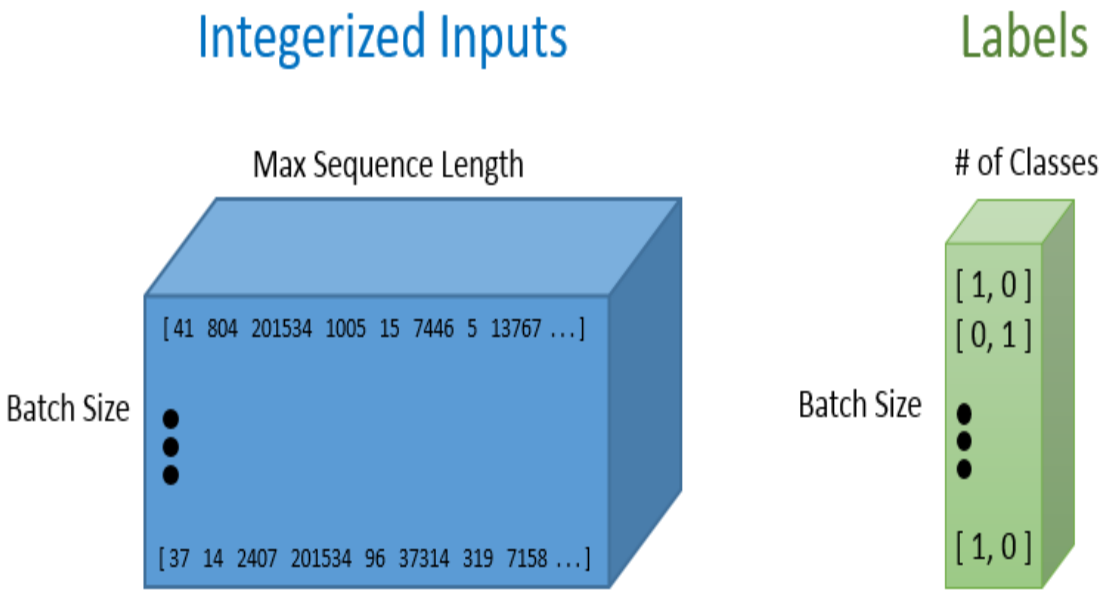
The training set that we are going to use is the Text dataset with individual lines of text in different text files present in the author folder. These text files are parsed through. WE can find out the average number of words per file and we can safely say that most reviews will fall under 250 words, which is the max sequence length value we will set.

Now we shall take the file and transform it into the ids matrix, once it is converted, we'll load in the training set and integrate it to get a matrix. This was a computationally expensive process, and we had to run it for a long time to generate finally a ids matrix.

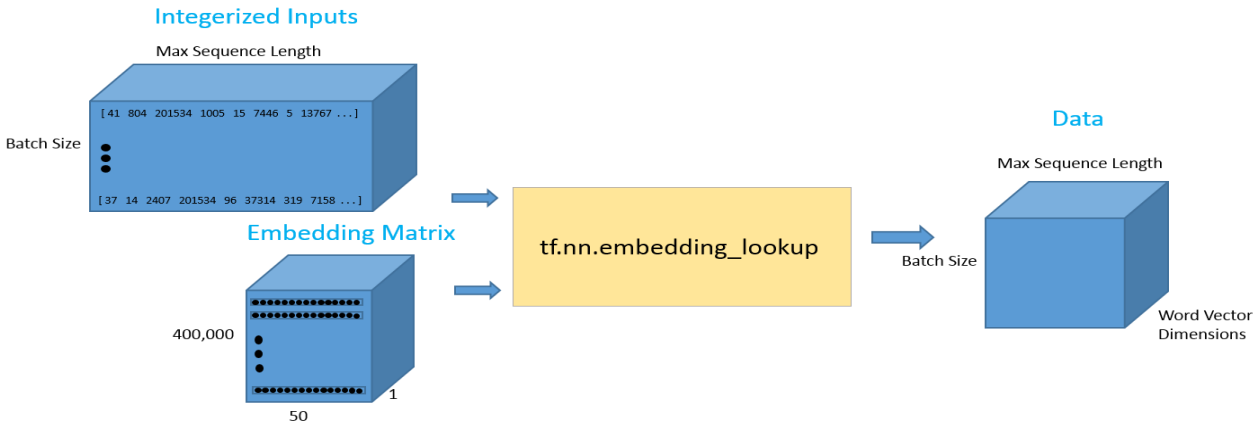
Now setting up the RNN model, we'll first need to define some hyperparameters, such as batch size, number of LSTM units, number of output classes, and number of training iterations.

As with most Tensorflow graphs, we'll now need to specify two placeholders, one for the inputs into the network, and one for the labels. The most important part about defining these placeholders is understanding each of their dimensionalities.

The labels placeholder represents a set of values, each either [1, 0] or [0, 1], depending on whether each training example is positive or negative. Each row in the integerized input placeholder represents the integerized representation of each training example that we include in our batch.



Once we have our input data placeholder, we're going to call the tf.nn.lookup() function in order to get our word vectors. The call to that function will return a 3-D Tensor of dimensionality batch size by max sequence length by word vector dimensions. To visualize this 3-D tensor, you can simply think of each data point in the integerized input tensor as the corresponding D dimensional vector that it refers to.



Now that we have the data in the format that we want, let's look at how we can feed this input into an LSTM network. We're going to call the tf.nn.rnn_cell.BasicLSTMCell function. This function takes in an integer for the number of LSTM units that we want. This is one of the hyperparameters that will take some tuning to figure out the optimal value. We'll then wrap that LSTM cell in a dropout layer to help prevent the network from overfitting.

Finally, we'll feed both the LSTM cell and the 3-D tensor full of input data into a function called tf.nn.dynamic_rnn. This function is in charge of unrolling the whole network and creating a pathway for the data to flow through the RNN graph.

As a side note, another more advanced network architecture choice is to stack multiple LSTM cells on top of each other. This is where the final hidden state vector of the first LSTM feeds into the second. Stacking these cells is a great way to help the model retain more long term dependence information, but also introduces more parameters into the model, thus possibly increasing the training time, the need for additional training examples, and the chance of overfitting..The first output of the dynamic RNN function can be thought of as the last hidden state vector. This vector will be reshaped and then multiplied by a final weight matrix and a bias term to obtain the final output values.

Next, we'll define correct prediction and accuracy metrics to track how the network is doing. The correct prediction formulation works by looking at the index of the maximum value of the 2 output values, and then seeing whether it matches with the training labels.We'll define a standard cross entropy loss with a softmax layer put on top of the final prediction values. For the optimizer, we'll use Adam and the default learning rate of .001.

Choosing the right values for your hyperparameters is a crucial part of training deep neural networks effectively. You'll find that your training loss curves can vary with your choice of optimizer (Adam, Adadelta, SGD, etc), learning rate, and network architecture. With RNNs and LSTMs in particular, some other important factors include the number of LSTM units and the size of the word vectors.

- **Learning Rate**: RNNs are infamous for being diffult to train because of the large number of time steps they have. Learning rate becomes extremely important since we don't want

our weight values to fluctuate wildly as a result of a large learning rate, nor do we want a slow training process due to a low learning rate. The default value of 0.001 is a good place to start. You should increase this value if the training loss is changing very slowly, and decrease if the loss is unstable.

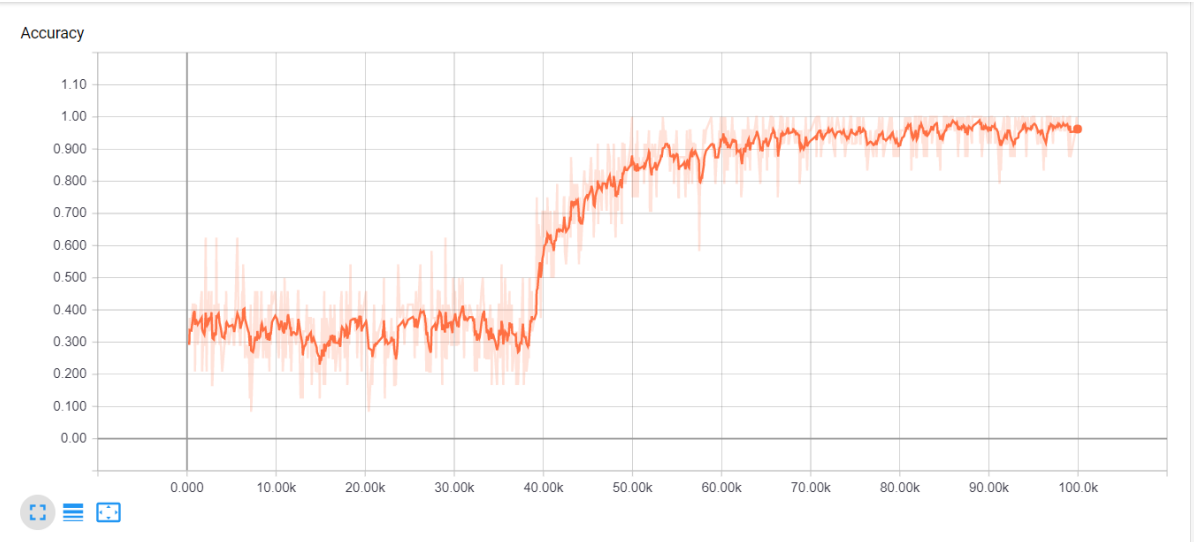
- **Optimizer:** There isn't a consensus choice among researchers, but Adam has been widely popular due to having the adaptive learning rate property (Keep in mind that optimal learning rates can differ with the choice of optimizer).
- **Number of LSTM units:** This value is largely dependent on the average length of your input texts. While a greater number of units provides more expressibility for the model and allows the model to store more information for longer texts, the network will take longer to train and will be computationally expensive.
- **Word Vector Size**: Dimensions for word vectors generally range from 50 to 300. A larger size means that the vector is able to encapsulate more information about the word, but you should also expect a more computationally expensive model.

Training Step

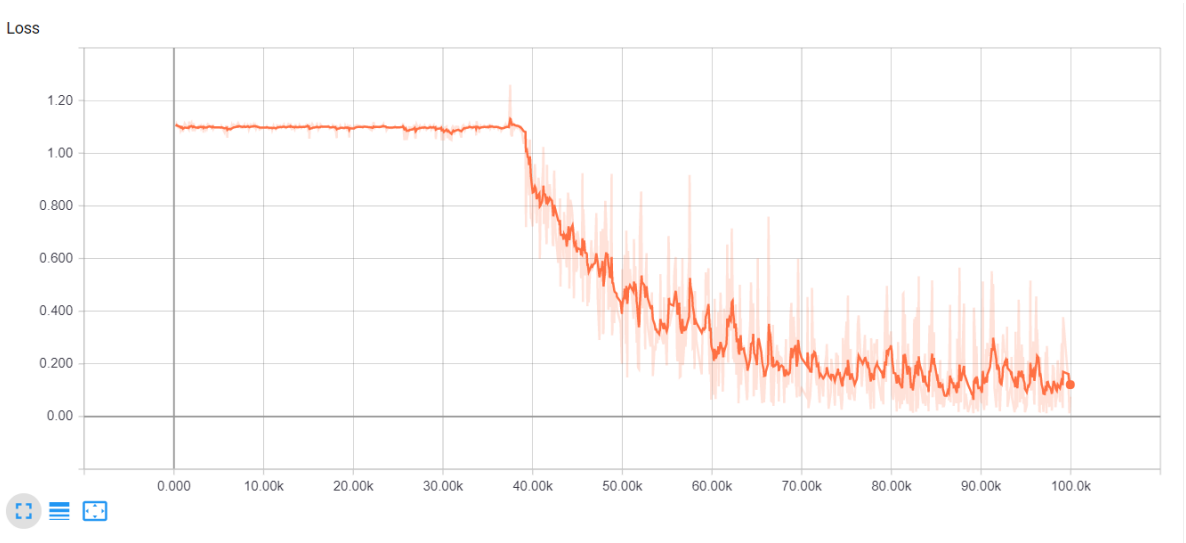
The basic idea of the training loop is that we first define a Tensorflow session. Then, we load in a batch of reviews and their associated labels. Next, we call the session's run function. This function has two arguments. The first is called the "fetches" argument. It defines the value we're interested in computing. We want our optimizer to be computed since that is the component that minimizes our loss function. The second argument is where we input our feed_dict. This data structure is where we provide inputs to all of our placeholders. We need to feed our batch of reviews and our batch of labels. This loop is then repeated for a set number of training iterations.

If you decide to train this notebook on your own machine, note that you can track its progress using TensorBoard. While the following cell is running, use your terminal to enter the directory that contains this notebook, enter tensorboard – logdir=tensorboard, and visit http://localhost:6006/ with a browser to keep an eye on your training progress.

Training Accuracy on Tensorboard:



Training Loss on Tensorboard



After loading the pre-trained model we try a random batch of the validation dataset on our model and get an avg accuracy of 68.33

```
INFO:tensorflow:Restoring parameters from models\pretrained_lstm.ckpt-85000

Accuracy for this batch: 66.6666686535
Accuracy for this batch: 79.1666686535
Accuracy for this batch: 66.6666686535
Accuracy for this batch: 62.5
Accuracy for this batch: 58.3333313465
Accuracy for this batch: 79.1666686535
Accuracy for this batch: 70.8333313465
Accuracy for this batch: 70.8333313465
Accuracy for this batch: 66.6666686535
Accuracy for this batch: 62.5
```

Conclusion

In the project, we have understood a deep learning approach to text classification. We looked at the different components involved in the whole pipeline and then looked at the process of writing Tensorflow code to implement the model in practice. Finally we trained and tested the model so that it is able to classify text based on authors.