MACHINE LEARNING

Predictive Text

Daniel I.	01FB15ECS086
Durga Akhil M.	01FB15ECS097
Ganesh K.	01FB15ECS104
Rahul R. Bharadwaj	01FB15ECS366

Contents

PROJECT REVIEW

Problem Statement:

To create a system which, given input as a part of a complete English sentence, can predict the next word in the sentence or generate the next sequence of words.

Dataset Details:

1. Corpus of Contemporary American English (COCA)

- About Dataset: The Corpus of Contemporary American English (COCA) is the largest freely-available corpus of English, and the only large and balanced corpus of American English.
- Attributes: We will be using a subset of COCA mainly from here. Hence, we will be using two, three, four and five-gram inputs to train the model each time.
- Instance count: Each of the n-gram datasets contains the following number of n-grams:
    - 2-gram: 1,020,386
    - 3-gram: 1,020,010
    - 4-gram: 1,034,308
    - 5-gram: 1,044,269
- This was the primary dataset for testing the goodness of fit for every other dataset as it was exhaustive. We also trained using this dataset for next-word prediction as well as generating text.

2. Wikipedia Corpus

- About dataset: This corpus contains the full text of Wikipedia, and it contains 1.9 billion words in more than 4.4 million articles.
- Attributes: Wikipedia Corpus contains Wikipedia pages scraped and dumped into text files. Since it is only for testing, we used random articles with a total of 2286765 words.
- Usage: The corpus is pre-processed using following steps:
    - Read all the lines of the input file.
    - Remove all unwanted characters and punctuations.
    - Remove lines with very few words.
    - Convert everything to lower-case.
    - Run a sliding window on the words
- This was the primary dataset with which we used to learn the model and apply goodness of fit test.

ML Techniques:

Chosen:

Following are the methods we shall employ and eventually compare the results to discover which is better for the given dataset

Markov Chain

A Markov chain is "a stochastic model describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event." A common method of reducing the complexity of n-gram modeling is using the Markov Property. The Markov Property states that the probability of future states depends only on the present state, not on the sequence of events that preceded it. This concept can be elegantly implemented using a Markov Chain storing the probabilities of transitioning to a next state. Also compared to RNNs its much faster to implement and provides a fairly good accuracy to complexity trade-off.

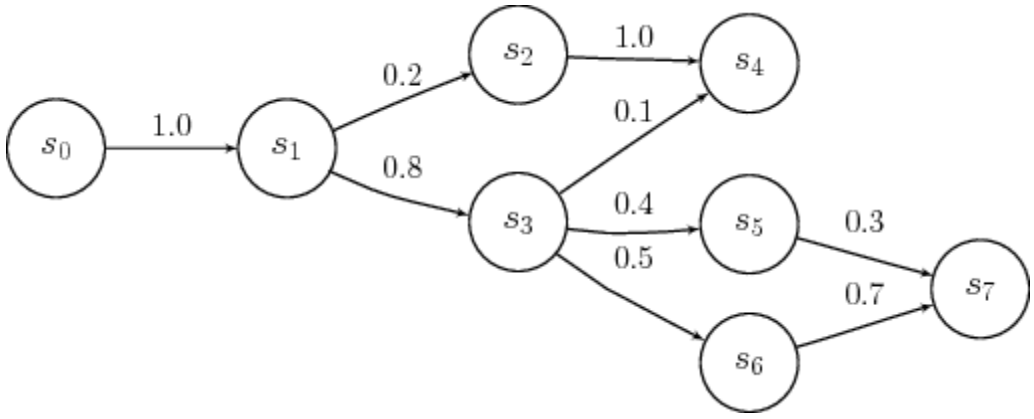

Figure 1 Example of Markov Chain

Recurrent Neural Networks:

RNN is a neural network which has recurrent connections, unlike feedforward networks. The major benefit is that with these connections the network is able to refer to last states and can therefore process arbitrary sequences of input.

- RNNs make use of sequential information. A traditional neural network assumes that all inputs (and outputs) are independent of each other.
- Since our project is to be predict text, it is crucial to know the information from previous states, i.e. the previous words to predict the next word.
- RNNs are called recurrent because they perform the same task for every element of a sequence, with the output being depended on the previous computations.
- RNNs thus have a “memory” element to capture information from previous calculations.

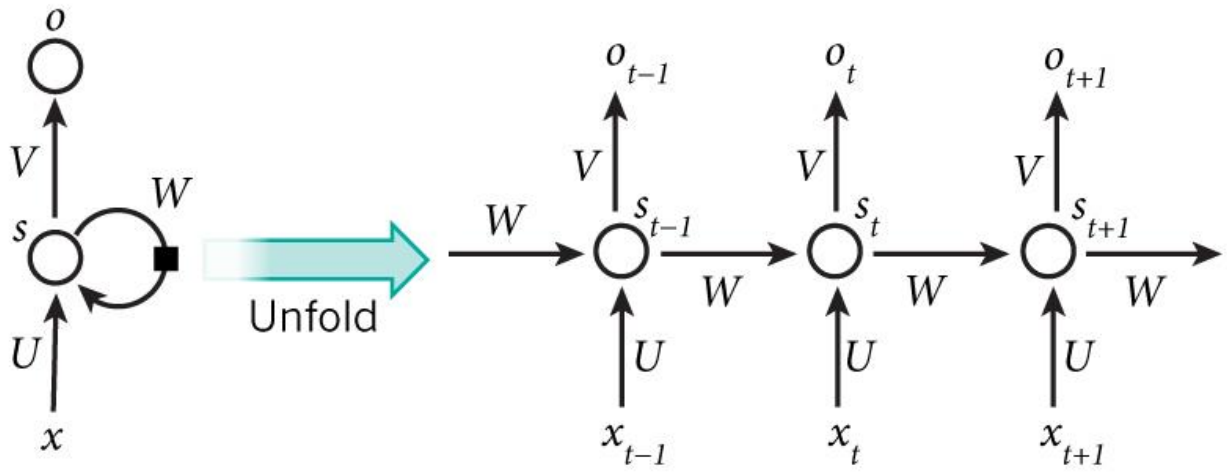

Figure 2 A typical RNN looks like one shown above. A RNN is unrolled (or unfolded) over time into a full network.

Need for LSTM and its advantages over RNN:

Two major problems with RNN: Vanishing and Exploding gradients.

- Vanishing Gradients: In traditional RNNs the gradient signal can be multiplied a large number of times by the weight matrix. Thus, the magnitude of the weights of the transition matrix can play an important role. If the weights in the matrix are small, the gradient signal becomes smaller at every training step, thus making learning very slow or completely stops it. This is called vanishing gradient.
- Exploding Gradients: refers to the weights in this matrix being so large that it can cause learning to diverge.

LSTM stands for Long Short Term Memory, is a special kind of RNN that learns long-term dependencies. The memory cell of LSTM is composed of four elements: input, forget and output gates and a neuron that connects to itself.

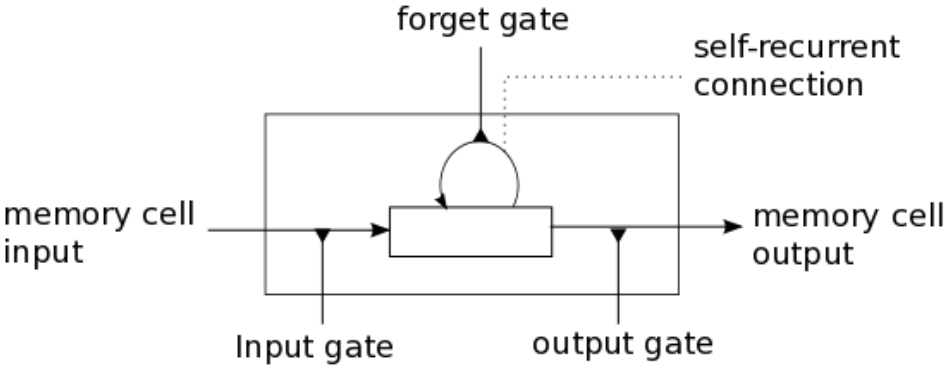

Figure 3 LSTM structure

Design Document:

Markov Chains

Consider bigram model: Input: `"I am Sam. Sam I am. I do not like green eggs and ham."`

Listing the bigrams starting with the word `I` results in: `I am`, `I am.`, and `I do`. If we were to use this data to predict a word that follows the word `I` we have three choices and each of them has the same probability (1⁄3) of being a valid choice.

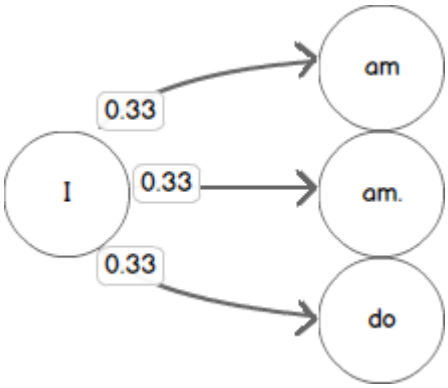

Figure 4 Modeling this using a Markov Chain results in a state machine with an approximately 0.33 chance of transitioning to any one of the next states.

We can add additional transitions to our Chain by considering additional bigrams starting with `am`, `am.`, and `do`. In each case, there is only one possible choice for the next state in our Markov Chain given the bigrams we know from our input text.

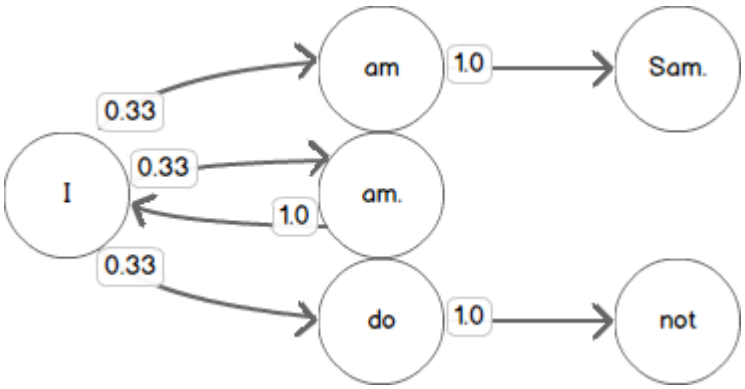

Figure 5 Each transition from one of these states therefore has a 1.0 probability.

Pseudo-Code (Python)

Create bigrams:

```
>>> s = "I am Sam. Sam I am. I do not like green eggs and ham."
>>> tokens = s.split(" ")
>>> bigrams = [(tokens[i], tokens[i + 1]) for i in range(0, len(tokens) - 1)]
>>> bigrams[('I', 'am'), ('am', 'Sam.'), ('Sam.', 'Sam'), ('Sam', 'I'), ('I', 'am.'), ('am.', 'I'), ('I', 'do'), ('do', 'not'), ('not', 'like'), ('like', 'green'), ('green', 'eggs'), ('eggs', 'and'), ('and', 'ham.')]
```

Markovian Chain: `I am Sam.`

```
{
    'I': ['am'],
    'am': ['Sam.'],
}
```

Add `Sam I am.`

```
{
    'I': ['am', 'am.'],
    'am': ['Sam.'],
    'Sam': ['I'],
}
```

Thus, give an input `'am'`, we get `'Sam.'`

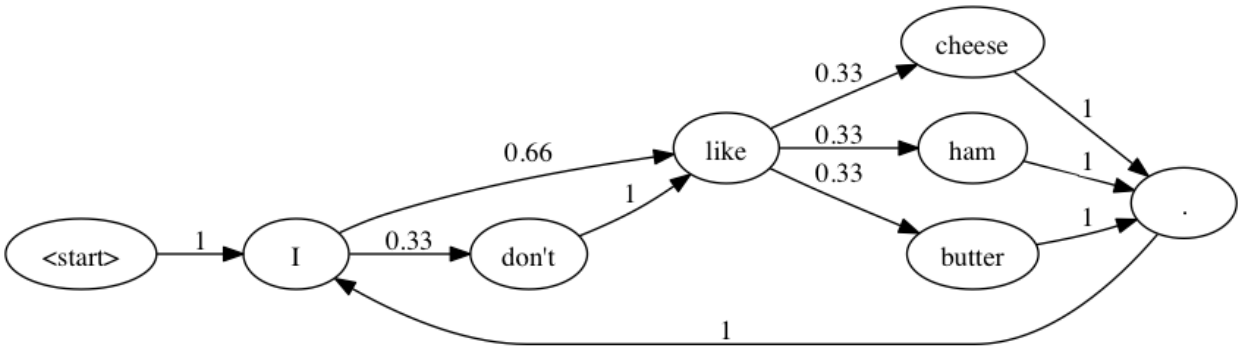

Figure 6 An example of fully modeled Markov Chain

Recurrent Neural Networks

Architectural Design

In order to train our recurrent neural network, we use a large text file (we have used the Harry Potter books) as input. This large text file undergoes preprocessing and is stored in the form of two arrays X and Y defined as:

X:

which is a three dimensional array of the form X[i, t, char_indices[char]] where i is the index of each sentence, t is the index of each letter in each sentence and char_indices[char] indicates the letter encountered. Therefore, X is like the given input matrix.

Y:

which is a two dimensional array of the form Y[i, char_indices[next_chars[i]]] where i is the index of each sentence and char_indices[next_chars[i]] indicates the letter which is encountered after encountering SEQUENCE_LENGTH (set to 40) characters of the sentence. Therefore, Y is like the expected output matrix.

We iterate through the text file at a step size of 3 characters, scanning through all characters, till SEQUENCE_LENGTH (store in list of sentences) and storing the next expected character after SEQUENCE_LENGTH (store in list of next_char). This is then used in the creation of the X and Y matrices.

Example:

For sequence length of 10 and step size of 3, the string

```
“Today is a Monday”
```

gets converted to:

```
Sentences = [“Today is a”, “ay is a Mo”, “is a Monda”] Next_chars = [“ ”, “n”, “y”]
```

X and Y are initially all initialized to False

Then, we update X and Y such that

```
1. X[0][0][‘T’] = True
2. X[0][1][‘o’] = True
3. Y[0][‘’] = True
4. Y[1][‘n’] = True
```

And so on.

BUILDING THE RNN MODEL

We use a Single LSTM layer with 128 neurons which accepts input of shape (the length of a sequence, the number of unique characters in our dataset). A fully connected layer (for our output) is added after that.

It has 57(number of unique characters) neurons and softmax for activation function:

```
1. model = Sequential()
2. model.add(LSTM(128, input_shape = (SEQUENCE_LENGTH, len(chars))))
3. model.add(Dense(len(chars)))
4. model.add(Activation('softmax'))
```

Our model is trained for many number of epochs using RMSProp optimizer and uses 5% of the data for validation:

```
1. optimizer = RMSprop(lr = 0.01)
2. model.compile(loss = 'categorical_crossentropy', optimizer = optimizer, metri
   cs = ['accuracy'])
3. history = model.fit(X, y, validation_split = 0.05, batch_size = 128, epochs =
    20, shuffle = True).history
```

RMSprop has been developed independently around the same time stemming from the need to resolve Adagrad's radically diminishing learning rates.

PSEUDO-CODE:

```
def prepare_input(text):
	x = np.zeros((1, SEQUENCE_LENGTH, len(chars)))

	for t, char in enumerate(text):
		x[0, t, char_indices[char]] = 1.
	return x

def sample(preds, top_n=3):
	preds = np.asarray(preds).astype('float64')
	preds = np.log(preds)
	exp_preds = np.exp(preds)
	preds = exp_preds / np.sum(exp_preds)
	return heapq.nlargest(top_n, range(len(preds)), preds.take)

def predict_completion(text):
	original_text = text
	generated = text
	completion = ''
	while True:
		x = prepare_input(text)
		preds = model.predict(x, verbose=0)[0]
		next_index = sample(preds, top_n=1)[0]
		next_char = indices_char[next_index]
		text = text[1:] + next_char
		completion += next_char
		if len(original_text + completion) + 2 > len(original_text) and
		next_char == ' ':
				return completion

def predict_completions(text, n=3):
	x = prepare_input(text)
	preds = model.predict(x, verbose=0)[0]
	next_indices = sample(preds, n)
```

```
        return [indices_char[idx] + predict_completion(text[1:] + indices_char[idx]) for idx in
next_indices]
```

Results Markov Chain:

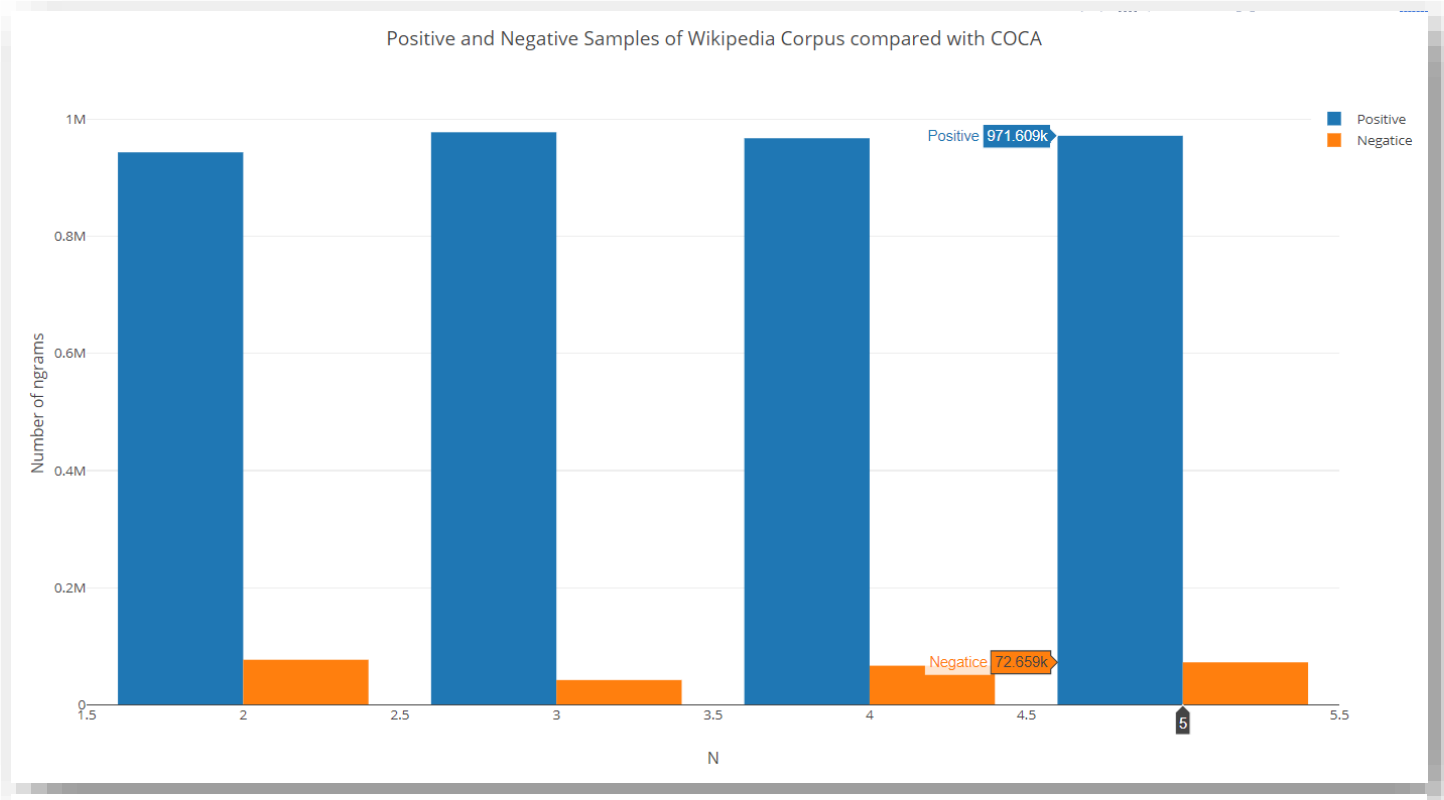

Figure 7

Table 1 Classification Accuracy and Error Rate for Wikipedia Corpus on COCA Dataset

N-gram	classification accuracy	Error Rate
2	92.44	7.6
3	95.84	4.16
4	93.53	6.47
5	93.042	6.958

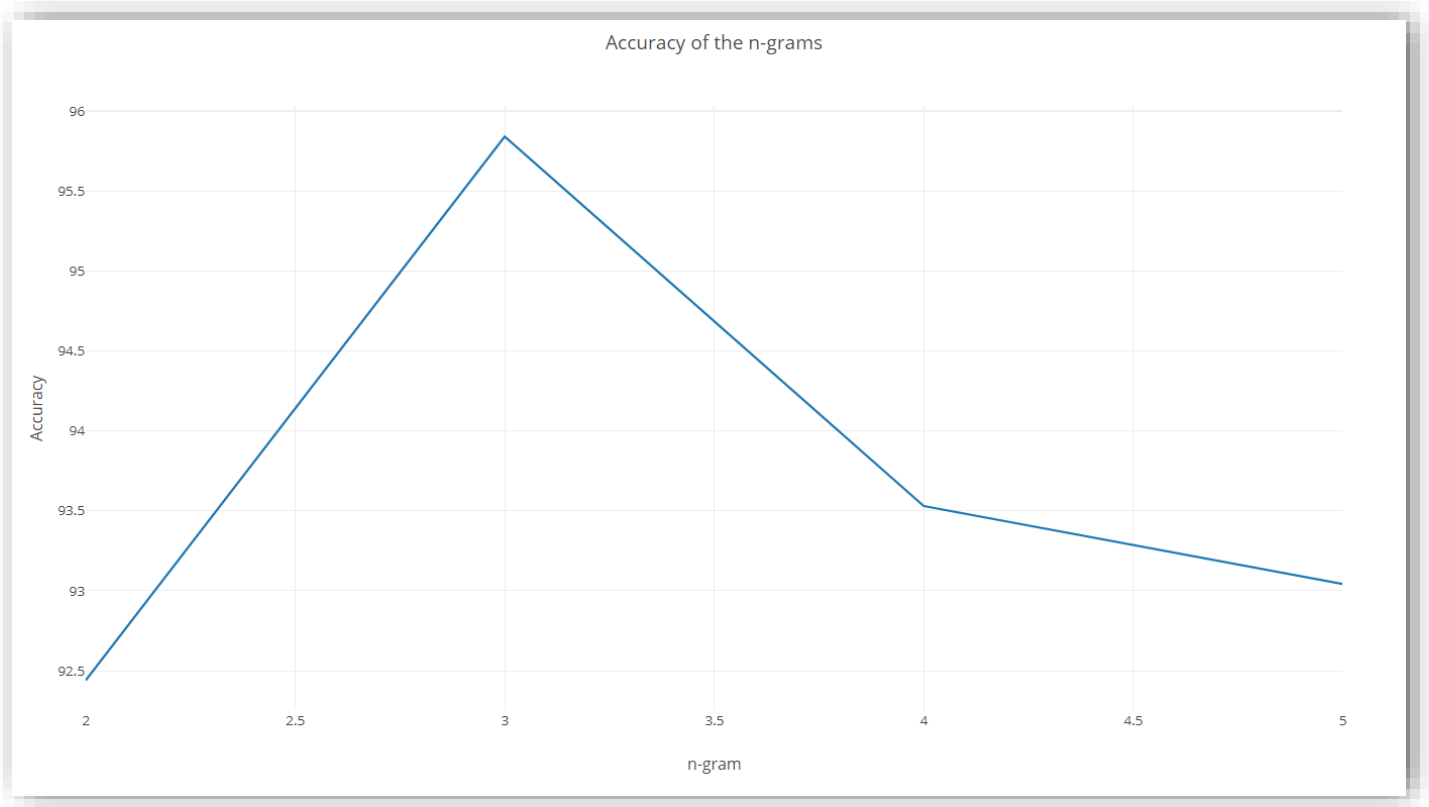

Figure 8

N-gram	w_acad	w_fic	w_mag	w_news	w_spok
2	67.89	62.83	66.74	68.91	67.84
3	72.97	69.27	68.76	71.98	70.29
4	71.67	66.54	67.45	70.27	69.92
5	70.23	67.78	67.23	69.89	69.02

Conclusion:

Thus, Markov Chains are very simple but effective tools for statistically modeling random processes. It is fairly simple to train compared to other models with similar levels of accuracy and very fast to query once learnt.

Although it is quite effective for predictive text, recent developments in Neural Networks has made Markov Chains the second options as models Like RNN have shown to give better results once trained properly.

Results RNN

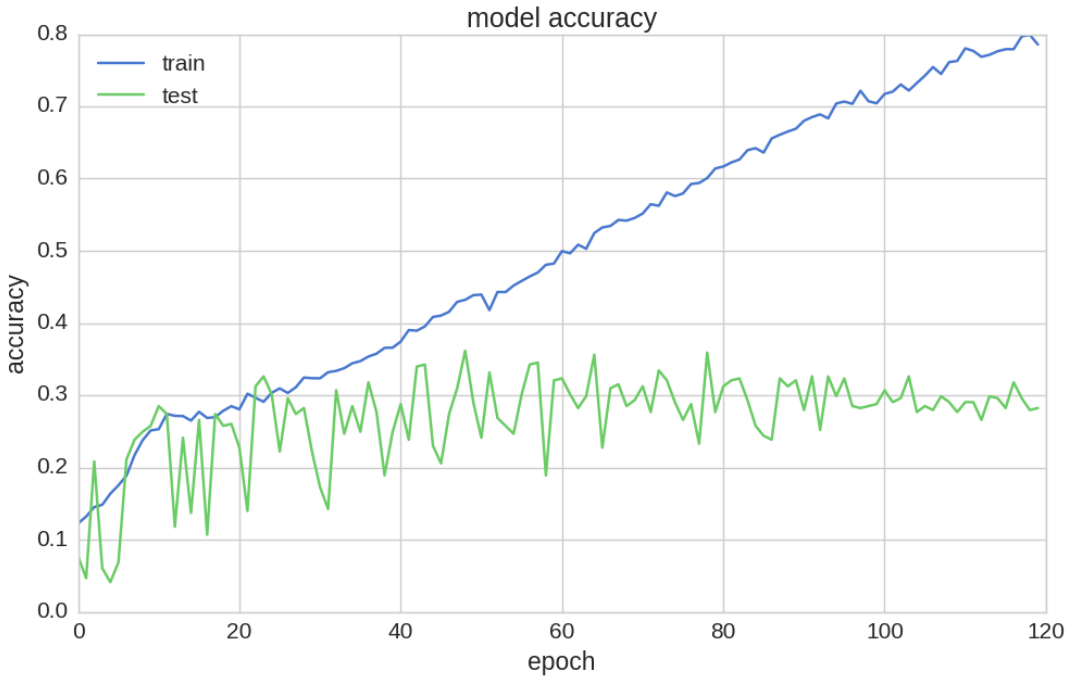

Figure 9

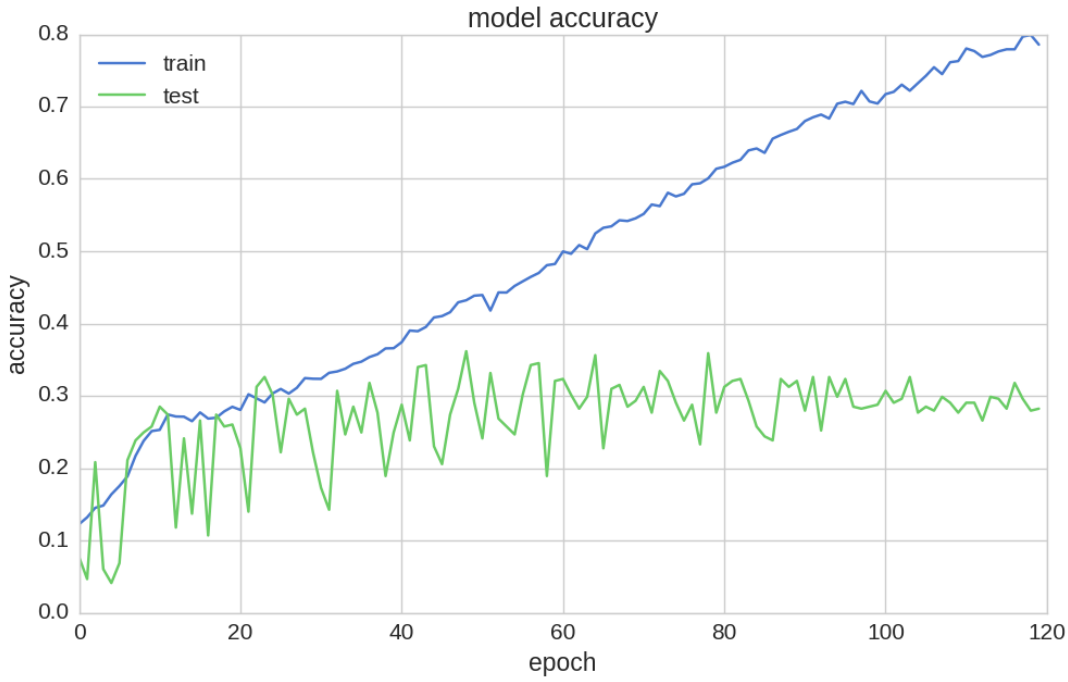

Figure 10

References:

- Language Model: https://en.wikipedia.org/wiki/Language_model
- Markovian Property: https://en.wikipedia.org/wiki/Markov_property
- Markov Chain: https://en.wikipedia.org/wiki/Markov_chain
- Dataset:
    - https://www.ngrams.info/download_coca.asp
    - https://corpus.byu.edu/coca/
- RNN: https://en.wikipedia.org/wiki/Recurrent_neural_network
- Implementations:
    - https://github.com/chfoo/tellnext
    - https://sookocheff.com/post/nlp/ngram-modeling-with-markov-chains/