2017

University of North Carolina Wilmington

Master of Science in

Computer Science and Information Systems

Proceedings

https://csbapp.uncw.edu/mscsis

ENHANCE HANDWRITING SCRIPT RECOGNITION WITH LANGUAGE MODELING

Yunkai Xiao

A Capstone Project Submitted to the
University of North Carolina Wilmington in Partial Fulfillment
of the Requirements for the Degree of
Master of Science

Department of Computer Science
Department of Information Systems and Operations Management

University of North Carolina Wilmington

2017

Approved by

Advisory Committee

Dr. Karl Ricanek	Dr. Douglas Kline

Dr. Curry Guinn, Chair

Accepted By

Dean, Graduate School

TABLE OF CONTENTS
(Insert Automatic Table of Contents)

ABSTRACT

Enhance handwriting script recognition with language modeling. Xiao, Yunkai, 2017. Capstone Paper, University of North Carolina Wilmington.

Current technology could already treat the handwriting-script recognition problem very well in certain domain. On other domains, such as recognizing damaged, faded, or stained scripts, the recognition rate are still not perfect. This paper intends to compare two language models (n-gram and LSTM network) that could potentially be used in enhancing the recognition rate for such domain by making prediction to the missing word. We found that with limited context given, the traditional n-gram model with Katz smoothing performs around 10 times better than the LSTM network. As a conclusion when the context given is limited, one should avoid using LSTM network on word prediction, and only use it when enough context is given. In the case of enhancing handwriting script recognition, this means to avoid using LSTM network when too many words are damaged or missing from the script.

LIST OF FIGURES

Figure Page

CHAPTER 1: INTRODUCTION

From more than half a century ago when people became aware of their needs of automatic reading systems, research and development on this topic have never died off. [3] Significant progress has been made to recognize machine printed text and even some domain of human handwritings in many languages, and to some specific domains, such as in machine printed addresses mail sorting of postal service, the problem has been considered solved. [3]

On the other hand, automatic recognitions of handwriting script as a generic problem still have not come out with a completed solution yet. The task that most researchers are working on is how to transform an off-line handwritten text's image into a machine-readable representation of the same text. The definition of off-line handwritten text will be discussed in the next chapter. Though in some very specific domain, such as bank check reading or address reading, people have come up with optimum solutions, such optimum solution on general handwriting recognition have not come up yet. [2] Part of the reason is because the tasks listed above all have a very small vocabulary and are single writer tasks. Generic handwriting recognition requires the system to operate under more complexed conditions, with very large vocabulary and unconstrained writing styles, maybe even under the distractions of different background [4]. Moreover, in some severe cases, such system has to deal with degraded scripts. Handwriting on script might partially fade out due to oxidizing, sun exposure, and other reasons. Part of the script may be damaged or missing, and there could even be stains from accidents covering up a portion of the script.

The task of recognizing handwriting scripts itself is already hard enough. Apart from the huge amount of words in natural language mentioned above, handwritings

normally have a large variety of character shapes [4], such as baseline orientation, slant angle, and size [9]. Not to mention that strokes along could have different size and width, and letters could be cursive and interconnected thus create a great variation in handwritings. [4][9]

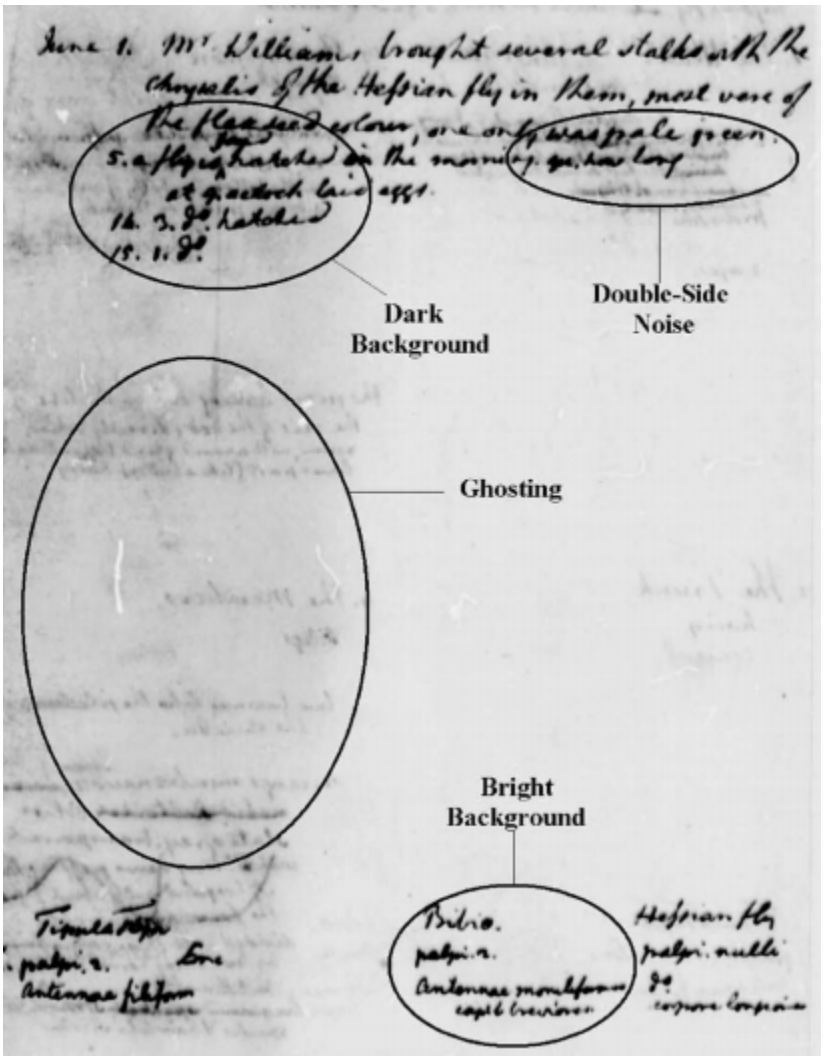


Fig 1. An example of a degraded historical document [20]

With all these difficulties, handwriting recognition programs may not be able to recover everything written on a script, which brings us to the topic of aiding this process with language models. Language model could aid the process in multiple ways, not

limited to predict words that could not be recognized and giving probability to words that are ambiguous to the recognizer.

In this paper, we are going to implement and compare the performance of two different language models, the n-gram language model and LSTM Recurrent Neural Network language model. Each of these models has its own characteristics and this paper is going to focus on specifically their ability to predict missing words.

By comparing the precision and recall of both model's performances on predicting missing words in sentences pulled from the billion-word benchmark corpus, this paper be able to reach a conclusion about the models' ability on aiding recognition to a certain degree.

CHAPTER 2: REVIEW OF LITERATURE REVIEW AND ANALYSIS

1.1 Handwriting Script Recognition

In this chapter, we are going to talk about how handwriting scripts are recognized by machines first, and then we are going to discuss where the enhancement could take place in this process and how we are going to enhance it.

Before people started to focus their research on recognizing handwriting script, mechanisms of recognizing machine printed scripts were developed first. Such mechanisms are called Optical Character Recognition (OCR) and are used on different occasions such as reading machine printed addresses in a mail-sorting machine with the speed of commercial processing system. [2]

As the demand of automatic recognizing of handwritings grows in the past century, research of automatic reading systems applied towards multiple domains have grown substantially and have achieved many goals. [3] A number of Industrial applications with near perfect recognition have been created. [2] Different approaches were created to work with different needs, and in most cases could be categorized into the following two type of handwriting recognition problems: On-line handwriting recognition and Offline handwriting recognition.

1.1.1 On-line Handwriting Recognition

On-line handwriting recognition is a technique that transforms pen trails into letters and words while people are writing. It relies heavily on using devices such as pressure sensitive pads or touch screens to capture motions of pen tips [1], and pen tip here is more of a generic concept of the tool or body part (such as a finger) that is used in writing. The process records motion the sequence of pen positions in two-dimensional space measurements as the digital representation of written texts [1], and most

importantly, the temporal information about the writing process. This information gives on-line handwriting recognition a great advantage. [2] In many circumstances that would hinder off-line recognition process, such as when neighboring letters overlap each other, the on-line recognition would be able to tell which part of the pen trail belongs to each letter thus being able to segment letters automatically. [1]

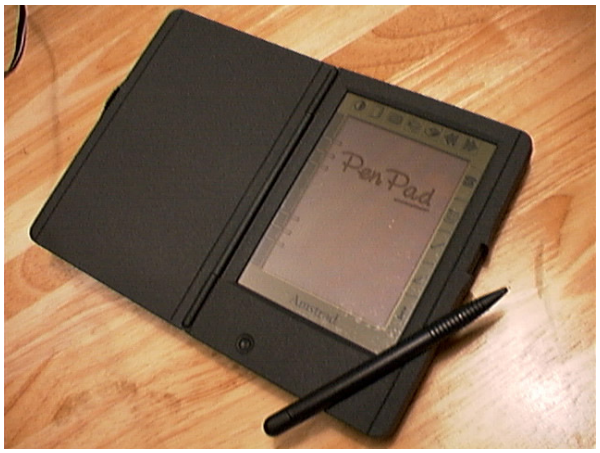


Figure 2. Amstrad PenPad (Author: Blake Patterson)

Besides the advantages listed above, on-line handwriting recognition made it easier to implement Markov model [3] [9] – a model that uses the change of states to recognize potential results. We will talk about Markov models later in the next section of this chapter.

Although on-line handwriting recognition has been implemented in many domains, including whiteboard reading [3] [9] and handwriting input on smartphones, we would not be able to implement it on history documents, especially those degraded scripts that are hard to recognize. Currently, there are ways to convert off-line handwriting records into on-line records, but could only achieve a certain level of accuracy and are not perfect. [10] In this paper, we will limit the scope to offline handwriting recognition.

2.1.2 Off-line Handwriting Recognition

In contrast to on-line handwriting recognition, off-line handwriting recognition processes documents that are digitally captured after they were completed and are then processed independently from the generation process. [1] This capture process is normally done using some sort of image creation tools such as a camera or a scanner. [3] The actual recognition of off-line data, normally in the form of images, usually involves using Markov models or Convolutional neural network. The fundamental prerequisite for the applicability of using Markov model is that the data considered can be represented in a linear sequence. [1] This is easy while doing on-line handwriting recognition, as the temporal progress is also captured in the data. When used in off-line recognition, some extra work has to be done to serialize off-line data. The next section will describe how recognition of off-line data is completed in steps.

2.1.2.1 Data Flow of Offline Recognition

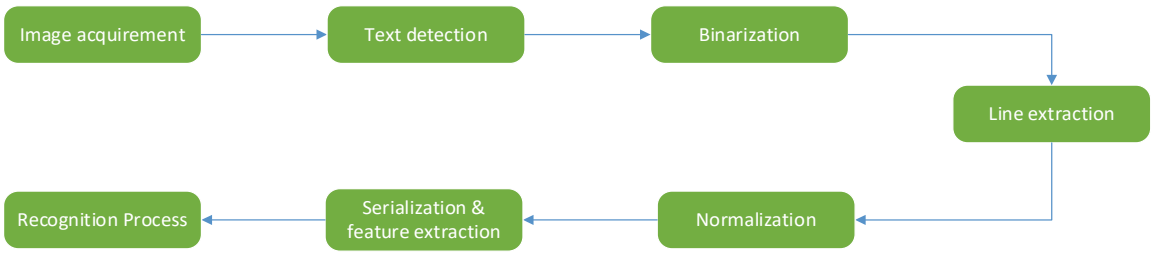


Fig 3. Data Flow of Offline Handwriting Recognition

Once an image containing text need to be recognized is captured, the first step is to extract the region of that image where text is located. Some common ways of doing this include using mathematical morphology operators, edge detection, and color information based extractions. Letters in text normally have much-intensive values that are significantly different from the background, thus made it possible for recognition. [11]

When areas of text are picked up from the image, some binarization would be applied to these areas. Binarization helps to separate text from background and to remove some degradations. This step involves global and local thresholding and could deal with different types of degradations such as faint characters, bleed through (text from the other side of paper showing up), and large stains. [5]

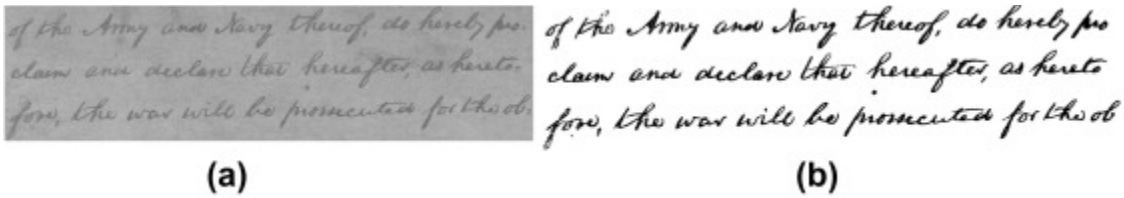


Fig 4. Binarization of Text: (a) Original image; (b) Post processing image [5]

The next step in the data flow is to extract lines of text from text block. Models, especially Markov models, are designed to recognize words in a sequence, and reconstruct this writing sequence in lines is a very intuitive thing to do. Besides, in the next few steps when text normalization is performed, it is performed in a line-to-line basis since the baseline orientation of text remains mostly consistent within the line.

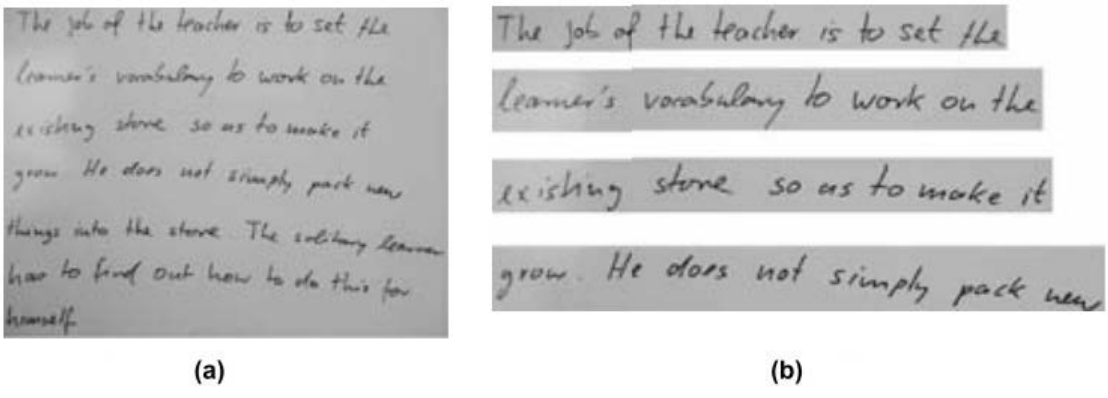


Fig 5. Line Extraction: (a) Text block; (b) Lines extracted from text block [3]

Normalization happens once lines are extracted from text blocks. This step normalizes many variations in handwriting, such as baseline orientation and slant. We

could compensate baseline orientation by tilting the line or rotating the image. The slant angle is represented as the tilt of individual handwriting letters, and is compensated by applying shear transforms to the line. [3] With a considerable amount of characteristics in terms of writing habits removed with normalization, the actual recognition process would be able to function with a much less complex model.

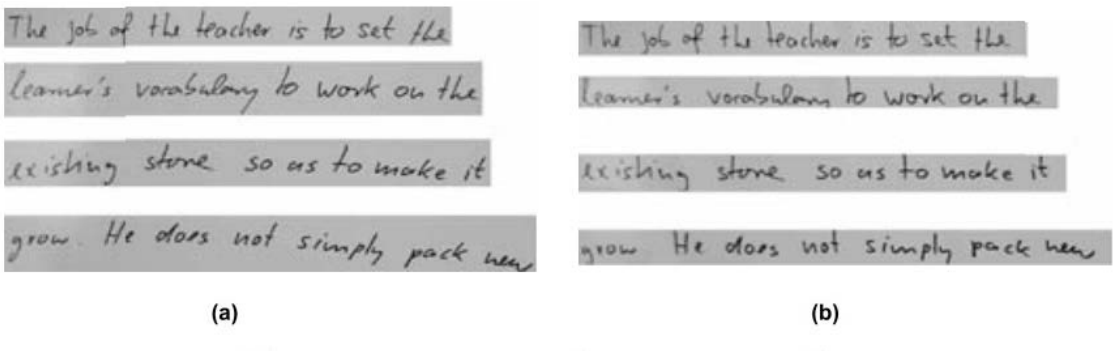


Fig 6. Normalization: (a) Before normalization; (b) After normalization; Notice that the variation in both of the baseline orientation and slant angle have been removed [3]

Finally, after preprocessing the recognition takes place. In the implementation of Markov models, the recognizer takes a serialization in, extract its features the make predictions on what it recognizes. [3] Although an off-line text image does not have its corresponding temporal data to track down the sequence of pen tip movements, people still find a way to serialize it. In order to generate a hypothetical timeline, a sliding window could be applied to the text line to mimic the sequence of changing local properties in time.

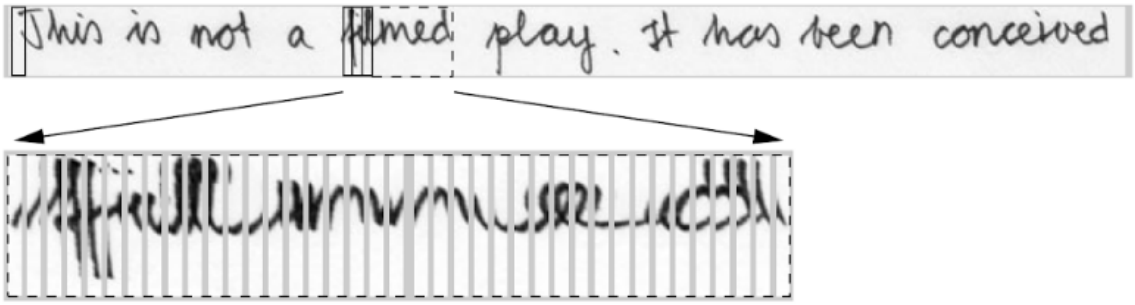


Fig 7. Serialization using sliding window [1]

A graph based handwriting recognition would then construct skeletons by adding nodes, endpoints and intersections then connecting them with edges. These skeletons could transform images into feature vectors in terms of dissimilarity space embedding. They are then compared with letter models to determine the dissimilarities, which is the minimum cost of edit operations needed (node or edge insertion or deletion, as well as node label substitution) to transform image read to model letters. [4] Then the input would be recognized into resulting letters based on minimum dissimilarities with Markov model.

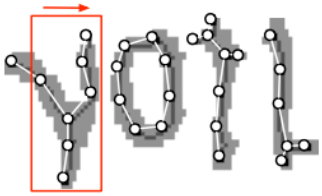


Fig 8. Skeleton with sliding window [4]

Markov model eliminates the need of segmenting individual letters since it processes a serial, or a sequence of items. This has made it much easier comparing with classical handwriting recognition systems as it takes fewer steps, and would reduce possible errors generated in letter level segmentation. [3]

This is where language models could step in and enhance the recognition process. Markov model and other newer technology such as neural network could indeed do some level of prediction on where the stroke is going thus make prediction on what a stained or damaged word could be, but they could only go so far, and the further they predict from known text, the less accurate they are going to be. Language models, on the other hand do not rely on the physical appearance of people's handwriting, instead they focuses on

context of the words itself. So once a unrecognizable word shows up on the script, one could always rely on language model's prediction to aid the recognition.

2.2 Language Modeling

Before moving forward in this section, we need to first define language model. A language model is a statistical model of word sequences. People use it to determine probabilities of words appearing in certain sequence, and of course, this could be used in predicting what would be the most likely word given a certain location in a word sequence. [12] The scope of this paper is comparing the two language models that could aid handwriting recognitions in predicting missing words.

They are the N-gram Language Model and the Long-Short-Term-Memory Recurrent Neural Network Language Model.

2.2.1 N-Gram Language Modeling

The N-gram language model is a type of probabilistic model, it checks on the probabilities of a sequence of words. Perhaps the simplest model of word sequence would be computing the chance of one word following any other word. If a corpus' vocabulary contains N distinguish words, then the possibility of one word following an existing word would be 1/N. This does not help a lot since only very little information has been fed into the model thus it could not distinguish different properties that different words held. A very intuitive way of improving this would be to give more information, such as words' frequency of occurrence. [12] For example if the word "cat" appeared 50 times in a corpus of 10,000 words, then the chance of that following any existing word is going to be 50/10,000. Can we improve the accuracy by giving more information? The answer is certain, by looking at the conditional probability of a word given its previous word. Suppose in the same corpus given above, the word "running" appeared for 60 times and

the frequency of word ”cat” appearing after that is 30 times, then given the word “running” the likelihood of “cat” to be the next word is 30/60. The reason we could achieve a much better result by doing this is that we have given the information of context, though just a very small piece, in this model.

Following this intuition, if statistic information of a longer sequence could be acquired ahead of time, better estimation for chances of a word being the last word of a specific sequence could be made. The length of this sequence names our model: if statistic information of every continuous n-word were given, this model would be called an “n-gram” model. This is a very effective model, though it might not be as accurate as a model recruiting every word appearing before the word we are trying to predict, it is good enough to make close estimations. [12]

In application, n-gram model always runs into a problem, which is there would almost certainly be some words that have not appeared in training corpus. This is a central issue in language model estimation called smoothing, which is the problem of adjusting the maximum likelihood estimator to compensate for data sparseness. [18]

One simple way of doing smoothing is to pretend that those words and the n-grams relevant to those words actually appeared in the data set by adding the total number of each possible n-gram by one. [19] This is called “add one smoothing”.

$$p_i^* = \frac{C_i + 1}{N + V}$$

This method may sound right, but is definitely unsound. Intuitively people would think the frequency of consecutive word’s occurrence would play a role in here, and then a modified technique has been proposed.

$$p^*_{(w_n|w_{n-1})} = \frac{\sum_{(w_{n-1}w_n)} + 1}{C(w_{n-1}) + V}$$

Among the enormous amount of smoothing techniques, Chen and Goodman (1996) found two that worked especially well, they are the average-count method and one-count method.

In this paper, with a huge existing training set, there is a better and easier to implement way of dealing smoothing problem – Katz smoothing or back off smoothing. The concept is very simple: if a word sequence cannot be found in higher order n-gram model, a search on the lower order model would be done instead, until the word is found.

2.2.2 Recurrent Neural Network Language Modeling

Apart from the basic n-gram model, some more advanced statistical model could be applied towards the problem. One of the recent models involves using recurrent neural network (RNN). The reason for introducing neural network language model in this study is that it has better adaptability on things that do not exist in training set.

The model we are going to discuss here is called "Long-short-term memory network", it avoids some of the problems that normal recurrent neural networks, especially long-term neural networks have. Traditional RNN is a type of neural network (NN) that preserves sequential information, or relationships within a sequence. [7]

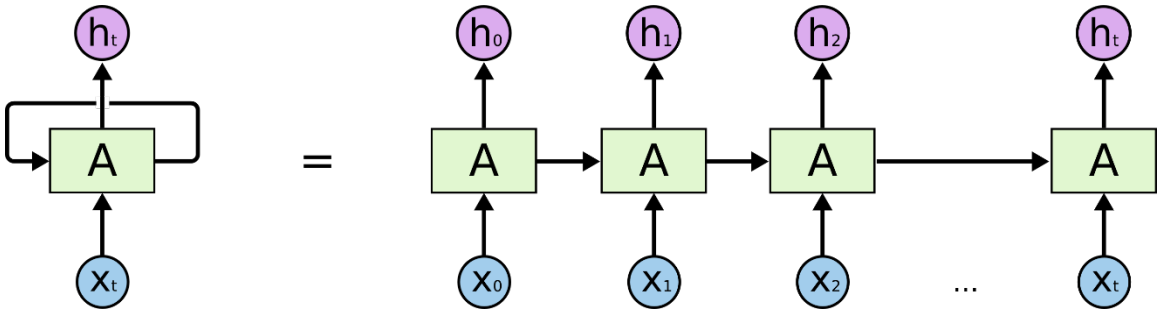


Fig 9. RNN [7]

Sometimes it would be beneficial to have some information from the recent past, and recent past is sufficient to make prediction. When trying to predict the word after

"We fed carrot to the snow white", it is not hard to come up with "rabbit" with no further information. However, in some cases such as "I grew up in France… I speak fluent French" in Olah's example [7], information in long-term context may be very helpful. Intuitively you would think RNN is perfect for this kind of situation, but the problem with it is as the gap between these two sentences grows, RNN's ability of connecting the information drops drastically.

When taking a closer analysis, people found theoretical and experimental results showing that gradient descent of an error criterion may not be adequate to train them for tasks involving long-term dependencies. [13] This is because RNN is trained by back propagation through time-steps, and as you can see in the previous figure when unrolling a RNN, each time-step is an equivalent of a layer in the feed forward network. If the RNN has a hundred time-steps, the gradient would vanish, as it would have been in a hundred-layer feed forward network. [26]

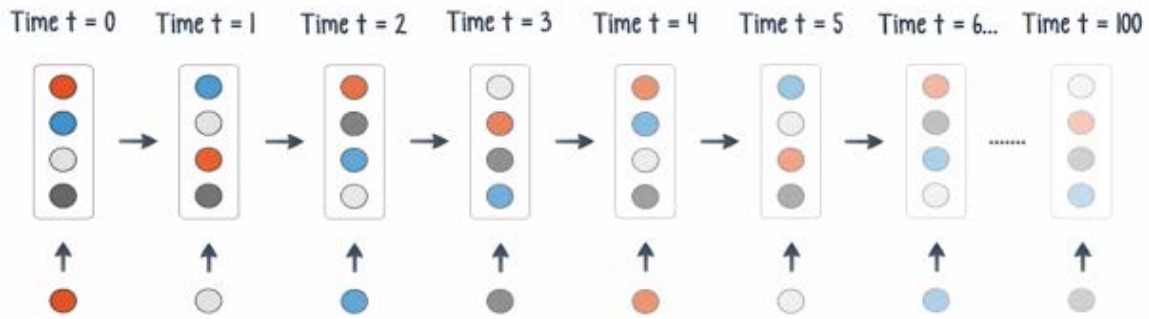
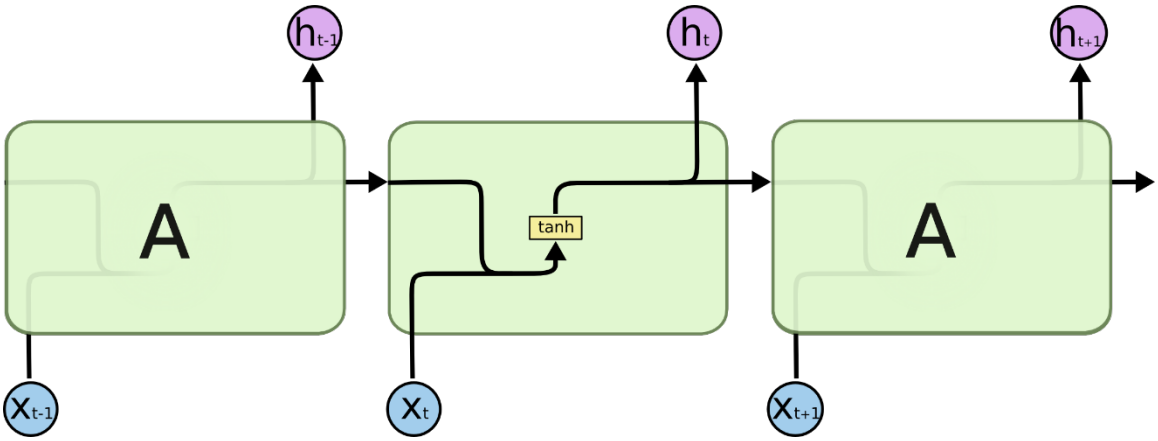
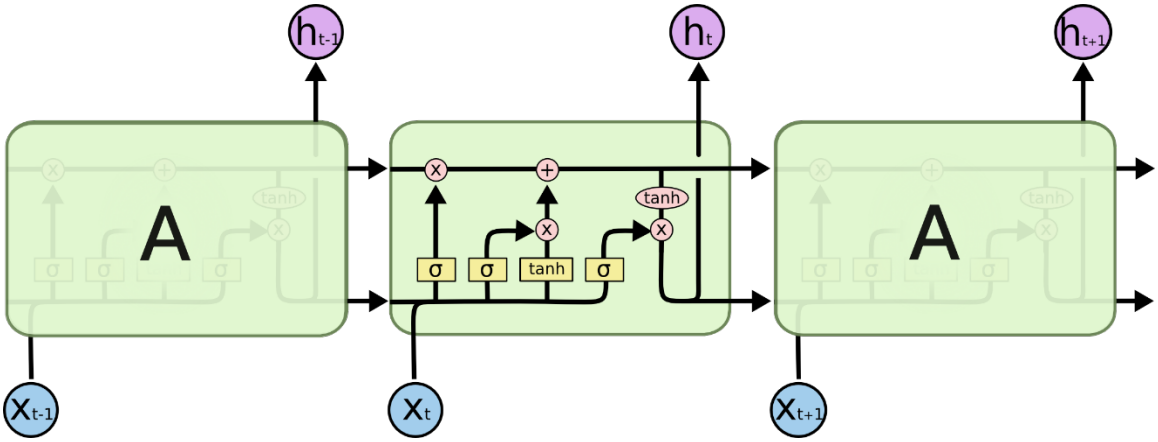


Figure 10. Gradient vanish through time [26]

Long-short-term memory network (LSTM) is a solution to this problem. Instead of having just one network layer, LSTM has four.



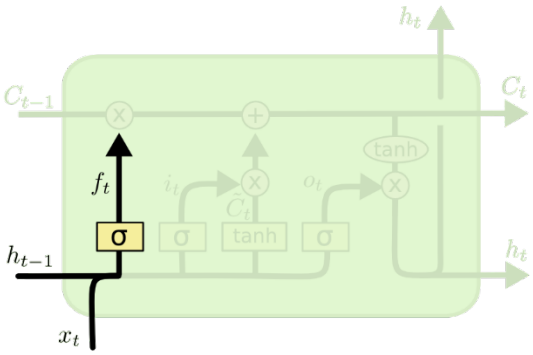
RNN



LSTM

Fig 11. RNN structure and LSTM network Structure [7]

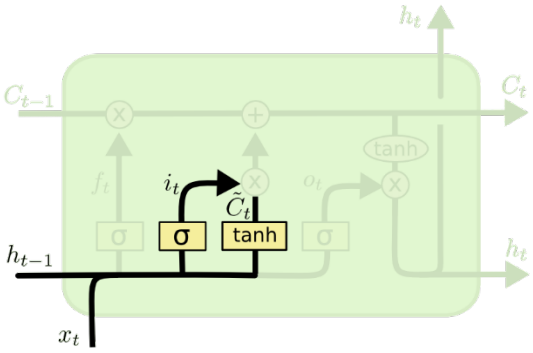
The idea around this is to have a separate channel (Channel C) that carries information (memory) necessary to the following content, instead of all of them, to the next cell. How does it decide which piece of information is useful to keep and which is not? What makes it "smarter" than regular RNN? The answer lays in the difference between these two networks – those extra network layers.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Fig 12. Forget layer [7]

Forget layer serves an important purpose together with other layers as a part of the progress on keeping necessary information in the memory channel. As you can see in the name, this layer focuses on forgetting information that is old and would not be useful, or even becomes harmful, on making predictions. Back to the sample that we gave above, if there is an "I stayed in Germany for 10 years" in between these two sentences "I grew up in France… I speak fluent _____", then perhaps the first piece of information about "grew up in France" would no longer be helpful towards predicting and should be forgotten. [7]



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Fig 13. Add and update layer [7]

Once unnecessary information has been removed from memory, we should consider adding new or necessary information to memory. As indicated by the name of this layer, it serves exactly this purpose. The input layer selects what need to be updated

merged with a select layer, which creates a vector of new candidate values to be added into memory. [7]

With the most up-to-date memory, combining with input of current cell, an output could be created here with the output layer. The current input makes decision about which aspect needs to go into output, and then draws them from adjusted memory. [7]

We have discussed about the structure of a LSTM model above, but there are still some other components need to work together with LSTM model to make it work. One of the most important steps before feeding data into the network is word vectorization. Vectorizing a word means extracting abstract information from the word to form a reasonably dimensioned vector that analytic models could process upon. Analytics has shown that word with similar meanings actually have similar features or dimensional representation in the vector space (see graph below). [14] This observation could also be implemented in other ways, such as using some offset technique with simple algebraic operations to predict where a word could be in the vector space. One example would be the result of vector("King") – vector("Man") + vector("Woman") is very close to vector("Queen")[15]

Vectorizing in neural network language models is the most important contributor to their adaptability. When given things not existing in training set, neural language model could still process them with a reasonable accuracy using things that are similar in vector space. An example for this would be given the example "Chickens are hatched from eggs", the network should be able to predict "Hens are hatched from ____" with "eggs" as well, since "hen" would have a very similar vector representation with "chicken". Because of this reason, we could expect a lesser performance drop in the

LSTM model comparing with n-gram model when the testing set is with different context of the training set.

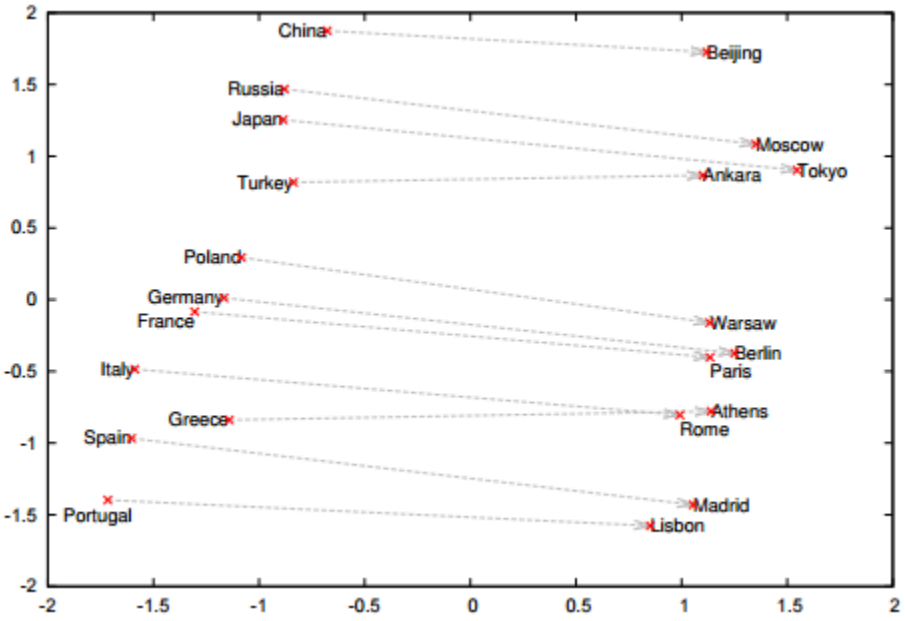


Fig 14. Vector space representations of countries and their capitals [14]

One way of doing vectorization is to use One-hot encoding; this creates a vector with the same size of vocabularies in the corpus. [6] It is obvious that this vector could sometimes be too big when you think about the size of any corpus and the amount of independent vocabularies within them. Luckily, the model we are going to refer to in this paper already has a pre-trained word-embedding network that could do this task once the checkpoint files are loaded.

CHAPTER 3: METHODOLOGY

3.1 Data Set

Any project working with statistical method would need data to support their analysis. Carefully picked dataset sometimes benefits more than perfectly built models, so spending effort on picking the most suitable dataset is crucial to analysis projects.

As in this paper, we are going to use the "One Billion Word Benchmark" [20] corpus. This corpus has been released in 2013, and it contains around a billion words with a vocabulary size of about 800,000 words. Most of the content is news data thus a better performance focusing the same area could be foreseen, in the same time we could study how much performance would drop when tested against other corpuses in different categories between two language models.

The data set is published by researchers in 2014 and is freely available to the public on the internet [20], thus made it very easy for any researcher to evaluate their language models.

A great advantage of using this corpus is that people have done extensive study and made a LSTM model that could be reused on this corpus. [21] As for n-gram language model, it is widely accepted that the amount of data and the ability of a model to accommodate large amount of training are crucial to its performance. [20] Besides, to achieve impactful results in domains like recovering unrecognizable content in scripts, language modeling techniques need to scale up to large datasets. It is reasonable to assume that with this amount of data the n-gram model would perform better and less biased towards specific articles. Moreover, when researching the performance of two language models, keeping a consistent training set between them is very important.

3.2 Implementing models

3.2.1 Building n-gram model

Before making any prediction, a language mode needs to be build. This process includes parsing raw text, cleaning and segmenting words, and record statistical data.

3.2.1.1 Parsing

Parsing is the process of turning raw text into statistical data models; this could be n-gram model, artificial neural network, and a lot more. In this article, we are going to build up our own n-gram model on the same piece of data that Vinyals and Pan was using for their LSTM model. This should allow us to compare two models on the same training set thus give a reasonable conclusion for the comparison.

Raw text are text data before any manipulation, for instance, it could be text scrapped from the internet, or digital books made available to the public. To parse them into language model, the parser must be designed to fit the pattern of the target language model. For instance, when building up the n-gram model in this paper, we are going to use a sliding window with the size of n. Every time the window slides, consecutive n words would be recorded then analyzed before getting stored in the n-gram model. Each individual word and punctuation would be treated as a token in this process, excluding "'s" and "s'" following words.

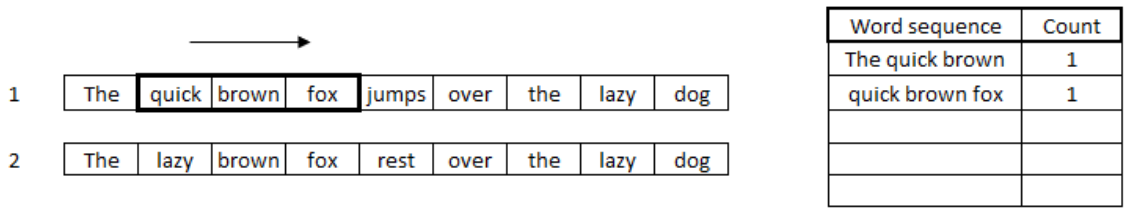
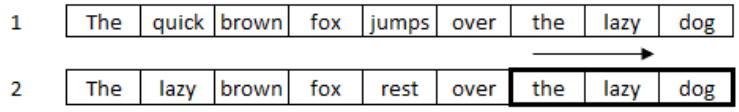


Figure 15. Sliding window with n = 3

In actual implementation, there is a lot more to consider. The dataset we are using in this paper has already been shuffled and stored in separate lines; the first step of cleaning and pre-processing is always to cut off format symbols. Such symbol may include indentation tokens, line wrap tokens, and a lot other tokens. Fortunately, Python has a built-in stripe function that we could call to deal with this problem. After stripping off format symbols, the sentence would be segmented into words. For languages like English, space between words could serve as a natural separator; program could simply determine the beginning and end of words with space and punctuation.

In this paper, we treat punctuations as words, however on many occasions there is no space between punctuation and the word before. With that said, some special rules must be applied to isolate them. The rule we are applying here are the following: Each of the segment in a sentence would go through an alphabet checker, if the segment contains non-alphabetical elements, then another checker would be applied. This second checker rules out the cases with number, "'s", and "s'", then pass it onto the punctuation separator. Here the unseparated punctuation will be cut off from the actual word then appended after it as an individual word.

The sliding window described in previous paragraph takes the cleaned word sequence and slides through it, generating tuples of size n. This tuple would be stored inside of a dictionary (hash table) in RAM temporarily for the moment. By the end of the process, we would have a dictionary containing all the word sequence appeared and its count. The figure below is a simplified representation of it.



Word sequence	Count
The quick brown	1
quick brown fox	1
brown fox jumps	1
jumps over the	1
over the lazy	2
the lazy dog	2
The lazy brown	1
lazy brown fox	1
brown fox rest	1
fox rest over	1
rest over the	1

Figure 16. A dictionary of word sequence and its counts after parsing

3.2.1.2 Record results

For reusing this data in the future, it must be stored somewhere. We have attempted a few methods for this purpose and they are:

3.2.1.2.1 Store in JSON file.

This method is very straight forward when thinking about it. A Python dictionary is easily convertible to Json file, and in-memory process is faster than other means of storage. Even if we would like to change into some other method in the future, it would be very simple to move on from it here.

While implementing the method, a few things come into realization. Python support converting data structures into JSON format storage file, while there is some specific requirement on this. One of the biggest problem is that JSON format does not support having tuple as key in dictionary. A quick work around of this problem is casting tuples into strings when storing, and evaluating them back into tuples while reading the stored file. At first, I have had the concern of the built-in evaluation function would miss treat cases such as quotes since they're wrapped with quotes as well when represented as string, but later find out cases like this could all be handled very well. The other problem is that some special character cannot be written into JSON file, after some try and

catches, we found that some Unicode 32 character may cause this problem, this could be fixed by specifically announcing file encoding while generating file descriptor by adding the parameter of “encoding = "utf8"”.

Another problem encountered in this parsing process is insufficient memory. Since dictionary in Python is a kind of hash table, when it expands, its size would double. Thus, while parsing for 4-gram and 5-gram dataset, the onboard memory always run out, even when I doubled it from 16GB to 32GB. This has inspired some thought for me, if data cannot be stored in RAM, maybe we could use some data management tool that offers query ability.

3.2.1.2.2 Store in a SQL database

A database fits that description very well. In this study, we are using MySQL 5.7 with mysqlclient as its Python connector. While loading n-gram data from JSON file to database, we found many junk data with long words, sometimes longer than 300 letters. A closer inspection showed that majority of these are web link or words sequences without spacing. This might be caused by the large quantity of web-scrapped data in this dataset, and could be removed in the process of loading. We’ve set up a rule to not load any word sequence that contains word that consists more than 45 letters, as 45 is the default maximum size for a VARCHAR field.

```
Part of this won't comply:  ('httpwwwrescuerootercomlocationsSanFranciscoBayNorth', '-', 'httpwwwrescuerootercomlocationsSanFranciscoBayNorthCA')
Part of this won't comply:  ('-', 'httpwwwrescuerootercomlocationsSanFranciscoBayNorthCA', '.')
Part of this won't comply:  ('httpwwwrescuerootercomlocationsSanFranciscoBayNorthCA', '.', 'httpwwwrescuerootercomlocationsSanFranciscoBayNorthCAaspx')
Part of this won't comply:  ('.', 'httpwwwrescuerootercomlocationsSanFranciscoBayNorthCAaspx', '.')
Part of this won't comply:  ('httpwwwdowjonescomsalesandtradingproduct', '-', 'httpwwwdowjonescomsalesandtradingproductelementized')
Part of this won't comply:  ('-', 'httpwwwdowjonescomsalesandtradingproductelementized', '-')
```

Figure 17. An example of words exceeding word length limitation

Our JSON data file for 2-gram has a size around 1GB and the data file for 3-data is around 6GB. To load something at this scale into the database, we have applied a few measurements towards the database. First, we turned off duplication check for the

database, which means each table would have no primary key nor unique key word. Then we have set “autocommit=0”, this setting controls whether the database management system would execute insert commands as soon as receiving them. When this setting is set to 0, we would be able to perform a “bulk loading” to the data base, in other words inserts to the data base will not be done but accumulated until a “commit” command is given. It is a common move to drastically cut down insert time. When looking at time spent during insert, the most time-consuming stage is I/O operation, every time data is inserted into the database, the database management system would need to pause the progress and wait for a confirmation. On the other hand, if these inserts are accumulated into one batch, there would only be one confirmation required at the end of this batch insert. While one might pose some concern about reliability using this method, since data is not committed onto hard drive until the end thus could be corrupted during the process. In lab environment, we could minimize this risk by closing all other unnecessary processes, and manually run checks after insert has committed onto the disk.

Once all the n-gram data has been committed into the database, we would like to add an index onto both tables to enable fast query. Being more specific, it would be word one for 2-gram table and both word one and two for 3-gram table. We will discuss 4 and 5-gram later in this article. For unigram table, we would only need it to be sorted based on frequency of word appearance, since that’s the only parameter we are going to refer to once our back off model backs to unigram. The database refuses to do this at first, throwing out duplication errors. Later when we exam the problem we found the dictionary we are keeping in the JSON file is case sensitive while MySQL database is not case sensitive.

Therefore, prior to creating the indexes, we did some de-duplication. De-duplicating a smaller database may not be so hard, but for large tables with the size of 1.6GB for 2-gram and 8.8GB for 3-grams respectively, one might run into some other issues. The way we de-duplicate tables is using a group by query on word sequence and sum up their frequency of occurrence while storing results onto a new table. Running this de-duplication script has raised another few problems; one is the loss of connection to the MySQL server during query. Queries for de-duplication is unlikely to finish in a short amount of time, so the first thing one would need to change is the DBMS connection read time out to a larger number. I did set this field to 24 hours and left it running over night, but all the de-duplication finished in a few hours – much shorter than a day. The other problem is that the total number of locks may exceeds the lock table size. Default innodb_buffer_pool_size for a MySQL database is 8MB, while cleaning up tables at this size would need more than that space for swapping and tracking data. As a workaround, I assigned a redundant amount of 2048MB for the innodb_buffer_pool_size and solved the problem. With indexes assigned to respective tables, queries could be performed in a reasonable amount of time.

3.2.1.3 Failure to capture 4 and 5-gram data

Regarding 4 and 5-gram data's capture, I've tried a few solutions, including handing off the work load of aggregating word frequency count to database using upsert; dumping data into database for every 100,000 lines parsed then merge results. Neither of this works, with the upsert solution took the database's entire buffer space while jamming the input API, and the second solution simply taking too much time to aggregate. I would describe a possible way to generate 4 and 5-gram data in chapter 5.

3.2.2 Using existing LSTM Language Model

The purpose of this article is to compare the performance of two language models that could possibly be applied onto enhancing handwriting recognition. In this article, we are going to use an existing LSTM model published by the Google Brain team as our LSTM model. Though the LSTM network posted in Vinyals and Pan's work has already measured its perplexity, our experiment would have to be done in a smaller dataset due to the inability to capture 4 and 5-gram data. To compare two models under the same condition, we would need to test the LSTM model with word sequence that is as short as we had applied to the n-gram model.

To run our own test on the pre-built model, we need to setup this published pre-trained model locally. To replicate the same environment as Vinyals and Pan were building the network; a few things need to be established first.

3.2.2.1 Setting up TensorFlow with GPU support enabled on Linux environment

The model that Vinyals and Pan made was built under some flavor of Linux system, and they offered a quick Bazel build solution for people who want to test their model. Bazel is an open source tool that allows for the automation of building software, but first one would need to setup TensorFlow under the operating system before building.

Setting up TensorFlow on Ubuntu is not quite easy as one may think. It is especially hard when we chose to use the GPU support enabled version. First, the hardware platform must be compatible with CUDA; Then TensorFlow requires a specific version of CUDA: at the time this paper is drafted, the current version of CUDA is CUDA 9.0, while TensorFlow requires CUDA 8.0. Moreover, Ubuntu 16.04 comes with an open source graphic card driver of version 378, and TensorFlow requires an official version of 375. Keep in mind that while installing CUDA 8.0, it is likely that the installation tool would help you rolling back your graphics card driver, but this will not

automatically enable TensorFlow for the system. It still would not pass later when testing the installation, while there might be a lot of suggestions online guiding user to install another version of the driver, one should not attempt to install anything different comparing with the preferred driver version recommended by CUDA's installer. One way of fixing this error is to check version numbers in the GPU driver's configuration file and make sure that number is in consistency with the number that CUDA installer offers. Some times this number needs to be manually changed after a number of driver installations and rollbacks.

Following the installation of CUDA (toolkit), another component called cuDNN needs to be installed. This is a necessary component to enable using GPU support in the neural network. Notice that cuDNN also has to match the CUDA version, in this case 8.0; one may try to look for it on NVidia's developer archive page.

Once CUDA has been set up, installing TensorFlow with GPU support is no longer a problem. A common practice here is to establish a virtual environment for python, this allows user to "active" several different python environments, which offers the capability to run several projects of different environment (version of python, packages) requirements simultaneously without interfering with each other. In this article, we are going to use Virtualenv to establish the virtual environment.

In this specific model, Vinyals and Pan offered a function for word prediction. Within the evaluation function there is a sample sub-function that one could use to test the network with a few words. This sub-function take a few inputs: the preceding words that user offers, the maximum length of the sentence one would like to generate, and how many times the user wants to run the same preceding word sequence for results. One important thing about using the model is, it would generate different reliable results each

time. When I took a closer look at the model's graph definition and extracted all 2933 nodes and operators, I found random generators on the LSTM initializers. The idea of incorporate randomness into the network is to create a natural representation of probabilities of word occurrences when there is not enough context. It also means later when we extract predictions we would need to run each word sequence for a number of times to get this representation.

3.3 Predict with language models

After setting up both language model, we started predicting words using them. The test dataset we have comes from multiple sources: the left-off data from the billion-word benchmark corpus, the news section from the Brown corpus and the humor section from the Brown corpus.

The reason we choose these corpuses is that the billion-word benchmark dataset "contains mostly news data" according to Vinyals and Pan, thus it would be reasonable to use something with similar and different context to test against it. Testing with the left-off data would give us an estimate on how well language models perform on the same dataset, while the Brown-news data gives the estimate on how well they would perform on data with similar background, and the Brown-humor data would tell how they perform over data with a very different context. As news-articles are normally very formal which is the opposite of humors.

3.3.1 Predicting with MySQL database

Predicting succeeding word with MySQL database is a relatively easy task, the Katz smoothing model uses a back-off mechanism to deal with word sequences that has not been recorded. To implement this model, we would need to be able to access all three ngram tables. The program would query the first two words in 3-gram table first and try

to find the third with a descending order on frequency, and when the first word cannot be find on the table, the program would back off to the 2-gram table. If that remaining word were not on the table again, the program would fall back to the unigram table to search for the most frequent. The top five most frequent appearing word in the unigram table with this training set is “the”, “.”, “,”, and “of”.

3.3.2 Predicting with TensorFlow LSTM network

On the other hand, predicting with the pre-trained LSTM network is harder and takes much longer. Part of the reason is what we have mentioned above – we would need to run the same few preceding words for a number of times to collect different possible results in a statistical way. Not to mention part of the code from Vinyals and Pan’s work needs to be modified to fit the purpose.

I have added a function to take in test datasets and run it through the network for multiple amount of time then collect results from it. Being more specific, we are running every preceding word sequence thirty two times to generate outputs and consider doing more experiments with larger number in the future.

CHAPTER 4: RESULTS AND DISCUSSION

We have used 1,000 sentence and collected results for the three datasets going through the MySQL database. Collected the most popular word, top three most probable words and top five most probably words. For the LSTM network, we have tested the left-out data on it and hope to test the other dataset in the near future.

Notice that both n-gram and LSTM network are statistical models, though performances may be different, methods of applying are the same. The advantage of applying LSTM network is that artificial neural networks tend to have better performance when encountering missing words, which makes perfect sense considering they are trained over vector representations instead of words. Comparing word predictions with two different model on the same dataset, we found that under the condition we have set for testing, traditional ngram performs significantly better comparing with LSTM network.

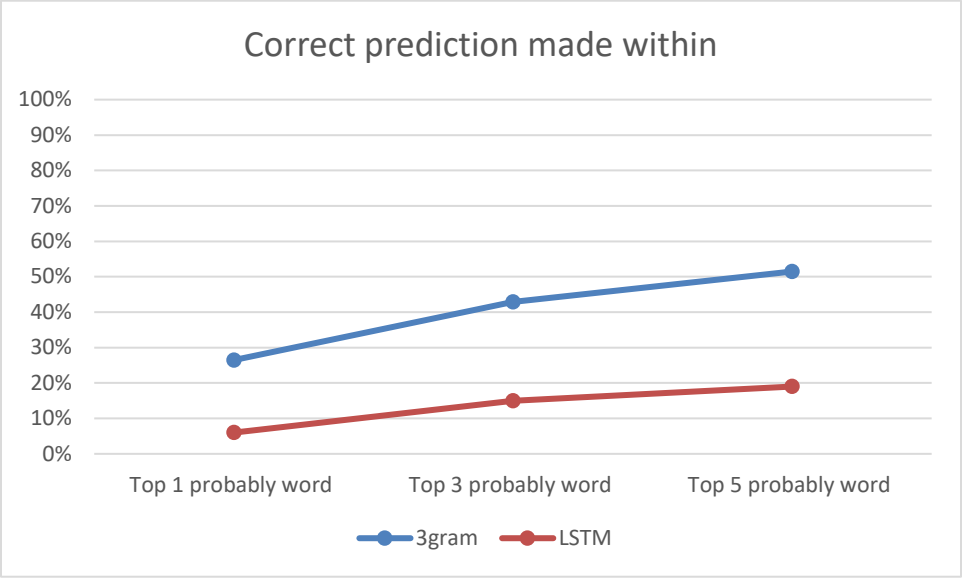


Figure 18. Percentage of correct prediction

As described in the previous chapter, the test dataset is generated from the held-out dataset [20] with randomly picked consecutive three words out of a thousand sentences. The first two words would be the preceding words given to both language models, while the third word checks against the result generated out of the models.

As we all could see from the graph above, the correct prediction made from the most probable word out of the ngram model yield around 26.5% while the LSTM model only made it to 0.63%. When we loosen up the criteria from the most probable to top three most probable words, ngram model's correct prediction rate rises up to 42.9% and though the LSTM network's correct prediction rate almost tripled, it still yields a very low score of 1.68%. Finally, when we further loosen the criteria to the top five most probable words, both models made the rate to 51.5% and 1.89% respectively.

This result shows that with very limited context (two preceding words), LSTM network cannot make accurate prediction comparing with traditional n-gram model. Part of the reason is that LSTM relies extensively on context, without enough preceding context fed into the network and with too many states randomly initialized, the network cannot pinpoint what is going to appear next. However, we have to mention that all of the result it has generated are perfectly readable and make sense to a human reader.

To research this problem, we tried to feed in more context to the LSTM network, and we did found a major performance improvement as it reached the context length in the training set mentioned in the model's creator – a sequence of words containing no less than 50 letters. [21] As can be seen in the figure below, there is basically no performance boost while under this 50-letter-threshold, but a major improvement when given this appropriate length of context.

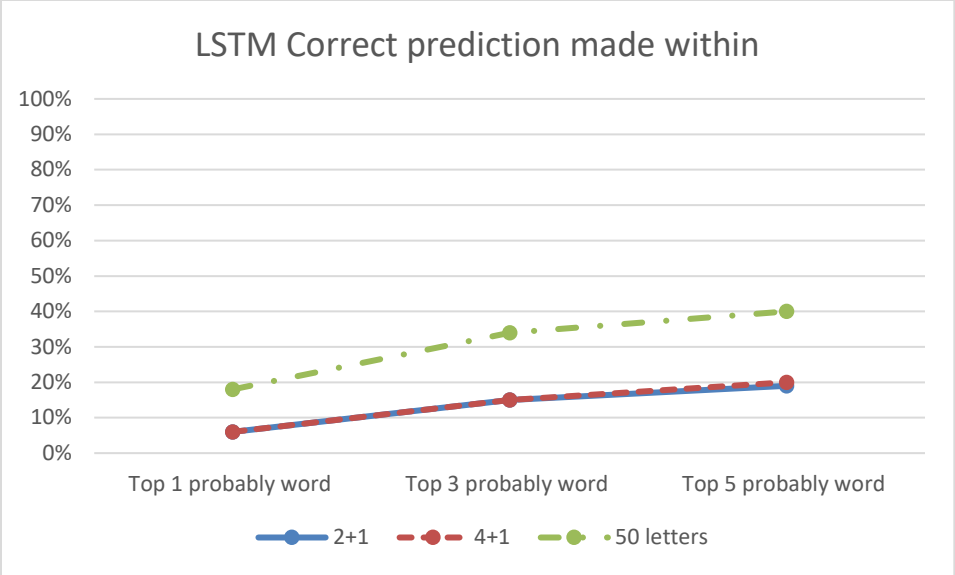


Figure 19. LSTM prediction accuracy with different length of context given

The other reason could be that the sampling size is not enough, while doing the experiment we have run each preceding word sequence thirty two times through the network. This might not be sufficient to form a representation that would review the true output population from the network.

On the other side, we originally assumed that ngram model would perform differently with different categories, that assumption might not be complete. Experimented with three different corpus, we found that the back off 3-gram model performed the best with the held-out dataset picked from the same articles with the same context from the training set. However when experimented with the Brown corpus, no matter which category we choose, the performance would not change too much.

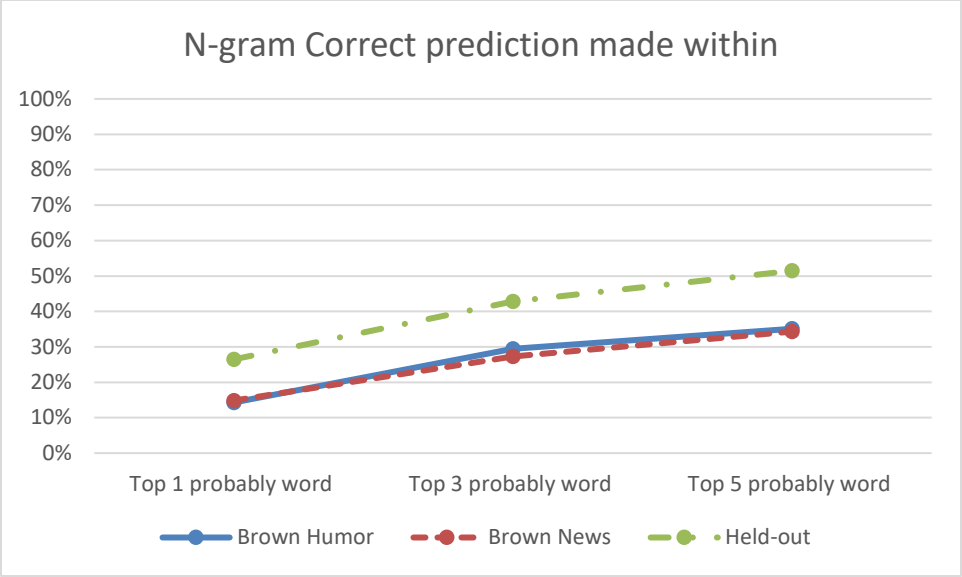


Figure 20. N-gram prediction accuracy with different corpus

Intuitively people would think ngram model would perform very differently with different article category, but our study shows that is not the case. The billion-word benchmark corpus consists mainly news data, but the n-gram model trained out of it performed similarly with the news category and the humor category of Brown corpus. These two categories are very distinct in terms of style; the news category is very rigorous and serious while the humor category is more relaxed. This has shown that though against intuition, the major influence on n-gram model's performance is the context of text instead of the category.

On the other hand, with LSTM models, we do experience content-related performance variance. Results from the Brown News category do perform slightly better than the Brown Humor category. This might be because although different words might be used in the Brown News test set, the structure and context of this set still have some

relevance to the Billion words training set.

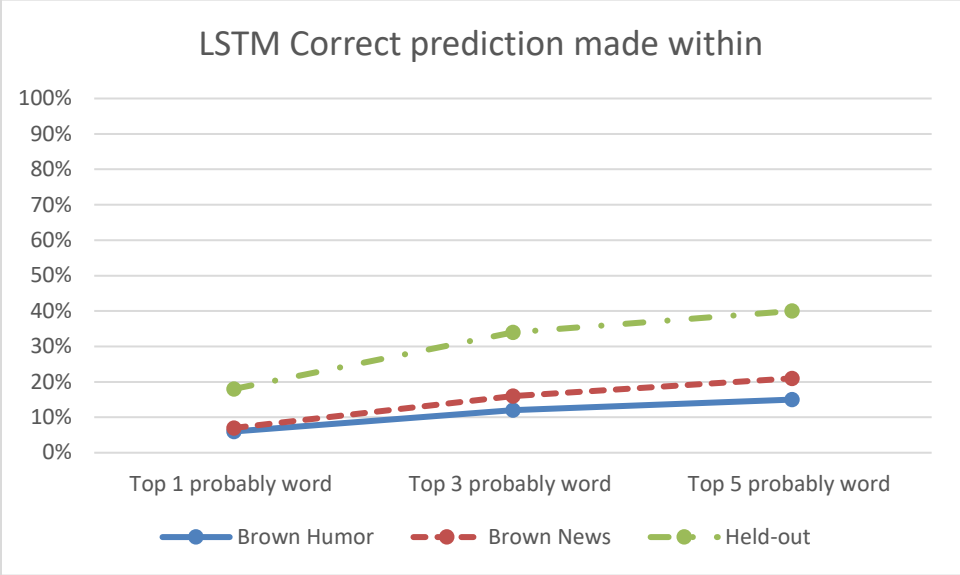


Figure 21. LSTM prediction accuracy with different corpus

CHAPTER 5: CONCLUSIONS AND FUTURE WORK-

5.1 Conclusions

In conclusion, language model could aid the handwriting recognition process, and different language model performs differently under different situations. LSTM network performs really well when given enough context [21], it would be more suitable to predict scripts that only have minor damages; While on the other hand, when large amount of the script is unrecognizable, traditional analytic models such as n-gram model would perform better. Language models always works better when trained with same context, which means if one is to predict words in a script written by a known person who has left sufficient amount of text that one could train his or her model with, it would always be nice to train with that data.

5.2 Future Work

5.2.1 Other smoothing techniques

In this article, we have used the Katz back off smoothing technique to deal with word inputs that are not recorded in the database. There are a lot other smoothing techniques that could be applied together with the n-gram dataset as well, and each technique would have its own benefit and drawbacks. Some future experiments could be done with these different smoothing techniques to measure the performance differences.

5.2.2 Use Google's ngram dataset

The Google n-gram data set is a dataset generated through Million Book Project. This is a project aiming at creating a free to read, universally accessible million book digital resource. The project is funded by NSF and is carried out with 15 partners in China, India and United States. [16]

Google worked on creating the n-gram dataset in 2007, and then revised it in 2012. Within the n-gram dataset, Google had parsed and recorded all appearances of word sequences up to 5-gram together with their frequency of occurrences within each year.

Predicting with existing ngram data from Google is very simple and straightforward. Since Google already made its n-gram data public in the format given above, then predicting words based on frequencies of occurrence become a problem of pinpointing the n-gram that possesses the largest frequency number in the history.

Since all of the data from Google is compressed in g-zip (.gz) format and stored in separate files starting with the first two letters of the first word in n-gram, the initial design is to parse that specific n-gram compressed file named after the first two character of target n-gram. While doing that, the program would sum up the frequency of possible n-grams in all years and compare them, before giving out the most frequent, which is the result.

This approach works, but have encountered a major problem in reality. While python does have a library that allows g-zip parsing, it is extremely slow. Especially when taken this into consideration: There are possibly hundreds of lines of temporal data written in the file for each n-gram – it has records for n-gram counts listed in years since the 1800s.

Some pre-processing work need to be done to deal with this problem, that is, before any parsing and searching takes place, all n-gram counts would be summarized, striping or implementing a quick summarize function towards temporal information in order to speed up query. Two different approaches has been tried, one is to load all data in its original structure to a MySQL database, the other one is to sum up each n-gram's

count in all years then store it in HDF5 format with indexing. Both method works, with SQL bulk loading described in the previous chapters or HDF5 file approach described in [Yarlett 2014].

With a dataset at this size, one should be able to observe some different results comparing with the ngram model trained with smaller model. What I am curious about is that whether this larger model would significantly improve prediction under different context, and whether performance would drop on the held-out dataset.

5.2.3 Use Scala and Spark to obtain 4 and 5-gram dataset

In this article, we have talked about the insufficient memory issue while parsing for 4 and 5-gram data. While this might be an issue on local machine, one could always use parallel computing technology. Spark is a framework built upon Hadoop, and it is built by the programming language Scala. Most people would choose either Python, Java or Scala while doing Spark programming but the nature of Spark made it very reasonable to use the same language it was built with to write Spark scripts.

Google offers a $300 budget for everyone who uses its cloud service at the time this paper is composed, and the cloud platform it offers could enable a very easy Spark setup, making it a perfect place to start with. Implementing word sequence counting is not a hard task with Spark, a master node would distribute sentences to its connected labor nodes while the labor nodes returns sequence and count. The distributed workload and Spark's collect method has lowered the requirement for each individual machine thus made it possible to handle data at this scale.

I would assume with longer input (more context), the LSTM network would perform better comparing to the current result. It is still in my mind to obtain 4 and 5-gram data and compare the n-gram model with LSTM model side by side again.

5.2.4 Train a LSTM model with limited context

The LSTM model used in this article is a pre-trained model on sentences, it would be reasonable to assume that when training with word sequence that is only five words long, it would perform somewhat differently. In the future, it would be something worth exploring to compare the performance with a model trained such way with the traditional n-gram language model.

5.2.5 Research how much word would be sufficient to build a robust language model

This article uses the one billion words benchmark dataset in its experiments. Although using more words in training would increase the robustness of a language model, our study did not show what would be the adequate size of a training set to construct a robust enough language model. We are uncertain if a billion words are too much for the purpose or too little, and when does the size of training set start to be irrelevant to the performance of language model.

We would like to research this topic in the future to find out a sufficient training set size for each language model to achieve adequate robustness.

.

REFERENCES

[1] Fink, Gernot A. Markov Models for Pattern Recognition From Theory to Applications. London: Springer London, 2014. Print.

[2] Zamora-Martinez, Francisco, et al. "Neural network language models for off-line handwriting recognition." Pattern Recognition 47.4 (2014): 1642-1652.

[3] Plötz, Thomas, and Gernot A. Fink. "Markov models for offline handwriting recognition: a survey." International Journal on Document Analysis and Recognition (IJDAR) 12.4 (2009): 269-298.

[4] Fischer, Andreas, et al. "A fast matching algorithm for graph-based handwriting recognition." International Workshop on Graph-Based Representations in Pattern Recognition. Springer Berlin Heidelberg, 2013.

[5] Ntirogiannis, Konstantinos, Basilis Gatos, and Ioannis Pratikakis. "A combined approach for the binarization of handwritten document images." Pattern Recognition Letters 35 (2014): 3-15.

[6] Coleman, Doug. "One-hot/One-of-k Data Encoder for Machine Learning." One-hot/One-of-k Data Encoder for Machine Learning. N.p., 25 Oct. 2012. Web. 21 Feb. 2017.

[7] Olah, Christopher. "Understanding LSTM Networks." Understanding LSTM Networks -- Colah's Blog. N.p., 27 Aug. 2015. Web. 21 Feb. 2017.

[8] "Recurrent Neural Networks | TensorFlow." TensorFlow. N.p., 8 Mar. 2017. Web. 25 Mar. 2017.

[9] Plötz, Thomas, and Gernot A. Fink. Markov Models for handwriting recognition. Springer Science & Business Media, 2012.

[10] Nguyen, Vu, and Michael Blumenstein. "Techniques for static handwriting trajectory recovery: a survey." Proceedings of the 9th IAPR International Workshop on Document Analysis Systems. ACM, 2010.

[11] Ezaki, Nobuo, Marius Bulacu, and Lambert Schomaker. "Text detection from natural scene images: towards a system for visually impaired persons." Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on. Vol. 2. IEEE, 2004.

[12] Jurafsky, Dan, and James H. Martin. Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. N.p.: Prentice Hall, 2000. Print.

[13] Bengio, Yoshua, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult." IEEE transactions on neural networks 5.2 (1994): 157-166.

[14] Mikolov, Tomas, et al. "Distributed representations of words and phrases and their compositionality." Advances in neural information processing systems. 2013.

[15] Mikolov, Tomas, et al. "Efficient estimation of word representations in vector space." arXiv preprint arXiv:1301.3781 (2013).

[16] Linke, Erika C. "Million Book Project." Encyclopedia of Library and Information Science: Lib-Pub (2003): 1889.

[17] St. Clair, Gloriana. "The million book project in relation to Google." Journal of Library Administration 47.1-2 (2008): 151-163.

[18] Zhai, Chengxiang, and John Lafferty. "A study of smoothing methods for language models applied to information retrieval." ACM Transactions on Information Systems (TOIS) 22.2 (2004): 179-214.

[19] Chen, Stanley F., and Joshua Goodman. "An empirical study of smoothing techniques for language modeling." Proceedings of the 34th annual meeting on Association for Computational Linguistics. Association for Computational Linguistics, 1996.

[20] Chelba, Ciprian, et al. "One billion word benchmark for measuring progress in statistical language modeling." arXiv preprint arXiv:1312.3005 (2013).

[21] Jozefowicz, Rafal, et al. "Exploring the limits of language modeling." arXiv preprint arXiv:1602.02410 (2016).

[22] Chen, Xie, et al. "Recurrent neural network language model training with noise contrastive estimation for speech recognition." Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on. IEEE, 2015.

[23] Bertolami, Roman, and Horst Bunke. "Integration of n-gram language models in multiple classifier systems for offline handwritten text line recognition." International journal of pattern recognition and artificial intelligence 22.07 (2008): 1301-1321.

[24] Peng, Fuchun, and Dale Schuurmans. "Combining naive Bayes and n-gram language models for text classification." European Conference on Information Retrieval. Springer Berlin Heidelberg, 2003.

[25] Breuel, Thomas M., et al. "High-performance OCR for printed English and Fraktur using LSTM networks." Document Analysis and Recognition (ICDAR), 2013 12th International Conference on. IEEE, 2013.

[26] Wang, Brian. "Recurrent Neural Nets." NextBigFuture.com. N.P., 06 Apr. 2017. Web. 01 Oct. 2017.

APPENDIX A

Building N-gram model with sliding window to Json:

```
import os, json

DIR = os.getcwd()

def gen(ngramNumber, cleanList):
    if ngramNumber > len(cleanList):
        return None
    returnList = []
    for index in range(len(cleanList) + 1 - ngramNumber):
        returnList.append(str(tuple(cleanList[index:index+ngramNumber])))
    return returnList

def lineCleaner(line):
    listOfWords = line.strip().split()
    returnList = []
    for word in listOfWords:
        if not word.isalpha():
            disregard = 0
            if "'s" in word or "s'" in word:
                disregard = 1
            else:
                for letter in word:
                    if letter.isdigit():
                        disregard = 1
            if disregard == 0:
                subword = ''
                for letter in word:
                    if letter.isalpha():
                        subword += letter
                    else:
                        if subword:
                            #append word
                            returnList.append(subword)
                        #append punctuation
                        returnList.append(letter)
        else:
            returnList.append(word)
    return returnList

def main():
    dirInfo = os.listdir(DIR)
    print(dirInfo)
    fileList = [fn for fn in dirInfo if '.en' in fn]
    print(fileList)
    for n in range(5,6):
        ngramCount = {}
        for file in fileList:
            lineCount = 0
            print("Start processing: " + file)
            fd = open(file, 'r', encoding="utf8")
            line = fd.readline()
            while line:
                lineCount += 1
                resultForThisLine = gen(n, lineCleaner(line))
                if resultForThisLine:
                    for item in resultForThisLine:
                        if item not in ngramCount:
                            ngramCount[item] = 1
```

```
                    else:
                        ngramCount[item] += 1
                if lineCount % 1000000 == 0:
                    print(lineCount, " lines parsed.")
                line = fd.readline()
            print(lineCount, " lines parsed. (EOF)")
            temp = open(file+'_'+str(n)+'gram_result', 'w')
            json.dump(ngramCount, temp)
            temp.close()

main()
```

Loading results in Json to MySQL database:

```
import os, json
import MySQLdb as mysql

def writeDBspecial(DBconn, dicToWrite, n):
    cursor = DBconn.cursor()
    cursor.execute('SET autocommit=0;')

    if n == 1:
        for item in dicToWrite:
            try:
                cursor.execute(
                    '''INSERT INTO ngram1 (w1, freq) VALUES (%s, %s)''',
                    list(eval(item)) + [dicToWrite[item]] )
            except:
                print("Part of this won't comply: ", item)
    elif n == 2:
        for item in dicToWrite:
            try:
                cursor.execute(
                    '''INSERT INTO ngram2 (w1, w2, freq) VALUES
(%s, %s, %s)''',
                    list(eval(item)) + [dicToWrite[item]] )
            except:
                print("Part of this won't comply: ", item)
    elif n == 3:
        for item in dicToWrite:
            try:
                cursor.execute(
                    '''INSERT INTO ngram3 (w1, w2, w3, freq) VALUES
(%s, %s, %s, %s)''',
                    list(eval(item)) + [dicToWrite[item]] )
            except:
                print("Part of this won't comply: ", item)
    elif n == 4:
        for item in dicToWrite:
            try:
                cursor.execute(
                    '''INSERT INTO ngram4 (w1, w2, w3, w4, freq) VALUES
(%s, %s, %s, %s, %s)''',
                    list(eval(item)) + [dicToWrite[item]] )
            except:
                print("Part of this won't comply: ", item)
    elif n == 5:
        for item in dicToWrite:
            try:
                cursor.execute(
                    '''INSERT INTO ngram5 (w1, w2, w3, w4, w5, freq) VALUES
(%s, %s, %s, %s, %s, %s)''',
```

```
                    list(eval(item)) + [dicToWrite[item]] )
            except:
                print("Part of this won't comply: ", item)
    # Would deduplicate later in DB, save time on constrain checking
    cursor.execute('COMMIT;')


def main():
    connection = mysql.connect \
            (
            host='localhost',
            user='root',
            passwd='woxingxiao',
            db='smallngram'
        )
    connection.set_character_set('utf8')
    fd = open('2gramjson','r',encoding="utf8")
    dic = json.load(fd)
    writeDBspecial(connection,dic,2)
    del(dic)
    fd = open('3gramjson','r',encoding="utf8")
    dic = json.load(fd)
    writeDBspecial(connection,dic,3)

main()
```

Test set generation and testing on n-gram model stored in MySQL:

```
from nltk.corpus import brown
import random, pickle
import MySQLdb as mysql



brownsent = brown.sents(categories = 'news')

def extract3(sentence):
    length = len(sentence)
    if length > 3:
        start = random.randint(0,length - 3)
        return sentence[start:start+3]
    else:
        return False

def lineCleaner(line):
    listOfWords = line.strip().split()
    returnList = []
    for word in listOfWords:
        if not word.isalpha():
            disregard = 0
            if "'s" in word or "s'" in word:
                disregard = 1
            else:
                for letter in word:
                    if letter.isdigit():
                        disregard = 1
            if disregard == 0:
                subword = ''
                for letter in word:
                    if letter.isalpha():
                        subword += letter
                    else:
                        if subword:
```

```
                            #append word
                            returnList.append(subword)
                        #append punctuation
                        returnList.append(letter)
        else:
            returnList.append(word)
    return returnList

def genSent(fileName):
    fd = open(fileName, 'r', encoding='utf8')
    sents = []
    line = fd.readline()
    while line:
        sents.append(lineCleaner(line))
        line = fd.readline()
    return sents

def genTest(sents,pickleFilename):
    sents = list(sents)
    numSents = len(sents)
    print(numSents)
    if numSents > 1000:
        random.shuffle(sents)
        count = 0
        validCount = 0
        dic = {}
        while validCount < 1000:
            words = extract3(sents[count])
            count += 1
            if words:
                dic[tuple(words[:2])] = words[2]
                validCount += 1
        fd = open(pickleFilename,'wb')
        pickle.dump(dic,fd)
        fd.close()
    else:
        print("Not enough sentences, you need at least 1000 valid sentences.")

genTest(brownsent,'brownNewsTestPickle')
genTest(genSent('news.en.heldout-00000-of-00050'), 'heldoutTestPickle')
genTest(brown.sents(categories = 'humor'), 'brownHumorTestPickle')

def testSQL(testPickle):
    fd = open(testPickle, 'rb')
    dic = pickle.load(fd)
    fd.close()
    connection = mysql.connect \
            (
            host='localhost',
            user='root',
            passwd='woxingxiao',
            db='smallngram'
        )
    connection.set_character_set('utf8')
    cursor = connection.cursor()
    top1Count = 0
    top3Count = 0
    top5Count = 0
    for item in dic:
        prediction = []
        cursor.execute('''SELECT * FROM final3 WHERE w1 = %s and w2 = %s ORDER
BY freq DESC''', item)
        results = cursor.fetchall()
        for result in results:
```

```
            prediction.append(result[2])
        if len(prediction) < 5:
            cursor.execute('''SELECT * FROM final2 WHERE w1 = %s ORDER BY freq
DESC''', item[-1:])
            results = cursor.fetchall()
            for result in results:
                prediction.append(result[1])
        if len(prediction) < 5:
            prediction += ['the','.',',','to','of']

        if dic[item] in prediction[:5]:
            top5Count += 1
        if dic[item] in prediction[:3]:
            top3Count += 1
        if dic[item] in prediction[:1]:
            top1Count += 1
    print(top1Count, top3Count, top5Count)

testSQL('brownNewsTestPickle')
```

Modified _SampleModel function in the LSTM model:

```
def _SampleModel(prefix_words, vocab):
    """Predict next words using the given prefix words.

    Args:
    prefix_words: Prefix words.
    vocab: Vocabulary. Contains max word chard id length and converts between
        words and ids.
    """
    fd = open('brownHumor100','rb')
    dic = pickle.load(fd)
    fd.close()
    count1 = 0
    count3 = 0
    count5 = 0
    count = 0

    for item in dic:
        item1 = item
        item = list(item)
        sess, t = _LoadModel(FLAGS.pbtxt, FLAGS.ckpt)
        count+=1
        print(count)
        possible = {}
        if item[0] != '<S>':
            item = ['<S>'] + item
        prefix = [vocab.word_to_id(w) for w in item]
        prefix_char_ids = [vocab.word_to_char_ids(w) for w in item]
        for _ in xrange(FLAGS.num_samples):
            targets = np.zeros([BATCH_SIZE, NUM_TIMESTEPS], np.int32)
            weights = np.ones([BATCH_SIZE, NUM_TIMESTEPS], np.float32)
            inputs = np.zeros([BATCH_SIZE, NUM_TIMESTEPS], np.int32)
            char_ids_inputs = np.zeros(
            [BATCH_SIZE, NUM_TIMESTEPS, vocab.max_word_length], np.int32)
            samples = prefix[:]
            char_ids_samples = prefix_char_ids[:]
            sent = ''
            while True:
                inputs[0, 0] = samples[0]
                char_ids_inputs[0, 0, :] = char_ids_samples[0]
```

```
                samples = samples[1:]
                char_ids_samples = char_ids_samples[1:]

                softmax = sess.run(t['softmax_out'],
                         feed_dict={t['char_inputs_in']: char_ids_inputs,
                                    t['inputs_in']: inputs,
                                    t['targets_in']: targets,
                                    t['target_weights_in']: weights})

                sample = _SampleSoftmax(softmax[0])
                sample_char_ids =
vocab.word_to_char_ids(vocab.id_to_word(sample))

                if not samples:
                    if vocab.id_to_word(sample) not in possible:
                        possible[vocab.id_to_word(sample)] = 1
                    else:
                        possible[vocab.id_to_word(sample)] += 1
                    break
        sess.close()

        sortedPossible = sorted(possible.items(), key=operator.itemgetter(0),
reverse=True)
        results = []
        for outcome in sortedPossible:
            results.append(outcome[0])
        print(results)
        if dic[item1] in results[:1]:
            count1+=1
        if dic[item1] in results[:3]:
            count3+=1
        if dic[item1] in results[:5]:
            count5+=1
        fd = open('humor100.txt','a')
        fd.write(str(item1)+'\n'+dic[item1]+'\n'+str(results)+'\n')
        fd.close()
    print(count1, count3, count5)
```