12-2017

Predicting the author of Twitter posts with Markov chain analysis

Daniel Freeman
University of Tennessee at Chattanooga, fgx556@mocs.utc.edu

University of Tennessee at Chattanooga

Department of Computer Science

Departmental Honors Thesis

Predicting the Author of Twitter Posts with Markov Chain Analysis

Author

Danny Freeman

Director

Dr. Craig Tanis

16 November 2017

Contents

1 Introduction

Given a set of text with known authors, is it possible to take new text, not knowing who wrote it, and correctly identify the author? One way to do this is to analyze the text using Markov chains. This research project will first attempt to answer this question using books available in the public domain. Using what is learned from trying to identify authors of books, the primary goal of this project is to identify the best way to guess the author of a post on the social media network Twitter using Markov chains.

Analyzing text requires changing it into a representation that can be processed by

a computer. The representation used in this research project is a Markov chain. A Markov chain is a process used for predicting future states in a state machine. This process has a set of states. The states are associated with a set of probabilities, each representing how likely it is the state machine will transition to the next state. These probabilities only depend on the current state of the Markov chain, with all information about the order of previous state transitions being lost with each transition to the next state.

Text can be represented as a Markov chain. This is done by tokenizing the text into either words or characters. The probabilities come from the likelihood that one token will follow the current one.

As an example, a Markov chain can be built using the characters from the title of this paper: "Predicting the Author of Twitter Posts with Markov Chain Analysis". Each character represents a state in the Markov chain. Here the letter "p" appears once in the title, and is followed by the letter "r". It follows that when the Markov chain is in the state "p", it has a 100% chance of transitioning to the state "r". More interestingly, the character "e" appears five times in the title. It is followed once by the letter "d", once by a space, once by the letter "r", and twice by the letter "s". When the chain is in the state "e", there is a 40% chance that it will transition to the state "s", a 20% chance for "d", a 20% chance for a space, and a 20% chance for "r".

Here is what the title looks like represented in a Markov chain as a graph.

```
{"p" {"r" 0.5 "o" 0.5}
 "r" {"e" 0.25 " " 0.5 "k" 0.25}
 "e" {"d" 0.2  " " 0.2 "r" 0.2 "s" 0.4}
 "d" {"i" 1.0}
```

```
 "i" {"c" 0.166666 "n" 0.333333 "t" 0.333333 "s" 0.166666}
 "c" {"t" 0.5 "h" 0.5}
 "t" {"i" 0.142857 "h" 0.428571 "w" 0.142857 "t" 0.142857 "e" 0.142857}
 "n" {"g" 0.333333 " " 0.333333 "a" 0.333333}
 "g" {" " 0.5 "e" 0.5}
 ;; ...
 "y" {"s" 1.0}}
```

The top level of the graph shows a state, followed by a second level representing other states that can be transitioned to, each with the probability that the Markov chain will transition from the state shown in the first level. States not represented in the second level of the graph have zero probability of being transitioned to from the state at the first level.

Modeling text this way allows for taking new pieces of text and calculating the probability that the new text could be represented in the original text's Markov chain. This is used to predict who wrote the new text without knowing the original author. Given a collection of Markov chains associated with different writers, the writer whose Markov chain yields the highest probability of generating the text from an unknown writer is assumed to be author of the unknown text.

1.1 Hy

This programming language used in this research project is an experimental LISP dialect called Hy that, at the time of this writing, is under active development on version 0.13.0. The language compiles to Python's abstract syntax tree, giving Hy the ability to call Python code and be called from Python code. This allows for popular libraries like the Natural Language ToolKit (NLTK), which is written in

Python, to be used from Hy [1]. Hy, unlike Python, provides access to a macro system that allows programmers to create new syntax rules for language and modify the order and manner in which Hy is evaluated. The tooling for Hy, some of which was written by the author while doing this research project, quickly evaluates code in small, easy to understand pieces. The tools provide rapid feedback by allowing code to be written and immediately evaluated in a Read-Eval-Print-Loop (REPL), without having to re-evaluate all the code in the project each time a change is made [2].

All code examples in this paper are presented in Hy. Being a LISP dialect, Hy has a very simple syntax. A brief overview of the language is given in a few examples here.

This first example demonstrates how a function is defined.

```
(defn function-name [param1 param2]
  function-body)
```

Like other LISPs, a function is invoked using this syntax:

```
(function-name 12 "cat")
```

Function calls can be nested as well, with the nested function call being evaluated first, and the result of that evaluation passed into the call to the parent function as an argument. Here, `function-one` is evaluated first, with the values `1`, `2`, and `3` being passed in. Then the result of the call to `function-one` is passed into `function-two`, along with the string `"test"`.

```
(function-two (function-one 1 2 3) "test")
```

Combining all of this, a simple working function can be defined and called like so:

```
(defn hello [name]
  (+ "hello " name))
(hello "world")
```

This defines a function called `hello` and calls it. The function call following the function definition in the example returns the text "hello world".

Functions can be nested, and a nested function will only be available within the local scope of the function it is nested in, much like a variable would be.

```
(defn function1 [x y]
  (defn nested [z]
    (+ x y z))
  (nested 1))
(function1 1 2)
```

This code evaluates to the integer 4. The function named `nested` is not available outside of `function1`. This allows for code inside a single function to be abstracted, while not unnecessarily making the abstracted code available to the rest of the application. Examples like this are seen all throughout this project.

Hy can interop with Python easily using dot notation. To access Python methods found in a module of defined on an object, the function can be invoked like so:

```
(.function-name myObject arg1 arg2)
```

This will be translated into the following Python code.

```
myObject.function_name(arg1, arg2)
```

Something else to note is that all dashes in Hy names (with the exception of the subtraction function), will be converted into underscores.

A handful of macros are used in this project. In particular, the threading macros (represented by the symbols `->` and `->>`) are used all throughout the code base for composing function calls in a way that is easy to read and modify. As a quick example, the thread-last macro `->>` will take in forms like so

```
(->> text
     (function1) ;; returns processed text
     (function2) ;; also returns processed text
     (list)) ;; breaks the text up into a list of characters
```

and rearrange it in to nested function calls like this

```
(list (function2 (function1 text)))
```

Adding another function to process the text in this case is much easier in the threading macro, compared to the expanded form that would have to be used if the threading macro was not available.

Hy uses Python's underlying data types, thus all decimal numbers used in this project are given by Python's floating point data type. The following Python code describes the limits of Python's floating point type, which uses IEEE double precision numbers on the machine this project was run on.

```
sys.float_info(max=1.7976931348623157e+308,
```

```
max_exp=1024,
max_10_exp=308,
min=2.2250738585072014e-308,
min_exp=-1021,
min_10_exp=-307,
dig=15,
mant_dig=53,
epsilon=2.220446049250313e-16,
radix=2,
rounds=1)
```

For more information on how this language works, refer to the Hy documentation at `http://docs.hylang.org/en/stable/` [2].

2 Related Techniques

Other researchers have tried identifying authors using a number of different methods. One common approach is using machine learning. It is usually applied by breaking text up into n-grams, which are ordered sets of `n` characters or words, and then comparing those n-grams to large bodies of work from known authors. A study done at Stanford University by Stanko achieved a success rate of 70% using machine learning [3]. This study also takes into account word lengths, vocabulary diversity, and syntax when trying to identify writers. One important advantage of this technique is that it takes into account previous information in a piece of text, whereas using a Markov process will only consider two words or characters at a time. The results for Stanko's method are more accurate when identifying longer pieces of text (novels) compared to small essays and papers.

Using Markov chains to examine texts has also been done in a number of studies. A study was done by Tran and Sharma in which they attempted to identify the language (not author) of text written in some European language by comparing the text to Markov chains created from known languages. They were able to achieve up to 98% accuracy in some cases. Their results found this method was very effective at identifying text containing between three and twenty words compared to Markov chains that were trained with up to 3000 words for each language [4].

A paper published by Khmelev describes a similar technique to identify authors using a Markov process. They were able to achieve an accuracy of about 74% when identifying authors of books comparing them to Markov chains [5]. Khmelev also applied this analysis to the Federalist papers, twelve of which the authors are disputed.

Where this research project will differ from similar studies is that the primary focus will be on identifying writers from Twitter. The other studies outlined here have all focused on formal writing from books and papers, which are longer and held to a higher standard than Twitter posts.

3 Methodology

3.1 Text Transformation and Normalization

Before any of the text can be processed in a meaningful way, it needs to be cleaned up. Not knowing what method for cleaning up text would work best, a combination of various methods of text transformation were used in this project. These methods of cleaning up text, referred to as text transformers, are small functions that take in a string of text, manipulate them in some way, then return the modified text.

Their simplicity allows them to be combined into a single transformation function that performs multiple transformations on the same piece of text.

One example is a lower case text transformer that takes in text and converts every character to its lower case variant (assuming one exists). In Hy, it simply looks like this:

```
(defn lower-case-transformer [text]
  (.lower text))
```

An example of a more complex text transformer is one that removes stop words and extra white space from text. Stop words are common words such as "the", "it", and "is". This transformer will remove stop words and also remove all line breaks, tabs, and extra spaces by replacing them with a single space character.

```
(defn strip-stopwords-and-whitespace-transformer [text]
  (import [nltk.corpus [stopwords]])
  (setv stops (.words stopwords "english"))
  (->> (.split text)
       (filter (fn [word] (not (in word stops))))
       (.join " ")))
```

The two transformer examples can be combined into a single transformer that converts text into a lower case variant, removes extra white space, and removes stop words.

```
(defn strp-stopwords-and-lower-case-transformer [text]
   (->> text
```

```
        (lower-case-transformer)
        (strip-stopwords-and-whitespace-transformer))
```

An arbitrary number of text transformers can be composed into a single transformer function with this function:

```
(defn compose-text-transformers [transform-functions]
  ;; Returns a new function that can preform multiple
  ;; transformations on a single text.
  (defn apply-transformer [text transformer]
    (transformer text))
  (fn [text]
    (reduce apply-transformer transform-functions text)))
```

To create a single transformer that applies N unique transformations to some text, they could be composed, then applied all at the same time using the `compose-text-transformers` function

```
(setv transform (compose-text-transformers [tr1 tr2 ... trN]))
(transform "some text")
```

There were other transformers used in this project in various combinations. Some of them include a transformer that replaces all Twitter handles with a single generic Twitter handle "@TwitterHandle", a transformer that replaces all URLs with a single URL "`https://t.co`", and one that strips out all punctuation from a piece of text. The reason for replacing all handles is to consider the Tweet without the mention of specific users. Unlike the use of proper nouns in books, the use of handles may not be consistent throughout a set of tweets. Users often engage in conversation with a

wide variety of different accounts, resulting in many unique and one-off transitions. Normalizing URLs may be necessary since Twitter will take URLs and shorten them with the prefix "`https://t.co/`" followed by a series of randomly generated characters that redirect to the URL a user has Tweeted. Instead of considering the random characters that were not really created by the user, a single link is used. In the Markov chains produced from text transformed this way, the transition from a word to handful of random URLs is replaced by the transition between a word and *any* URL.

Using these various text transformers and the ability to easily compose them, different experiments were run using different combinations of transformers in order to see which transformation methods produced the most accurate results. Hy makes it extremely easy to do this, since an experiment can be run with a single function that takes in different combinations of a composed transformer function as an argument. Running a variety of experiments can be done with a single function, and the way it is run (in terms of text transformation) can be altered by simply calling that experiment function with a different text transformer. The following code provides a quick demonstration of this.

```
;; Run an experiment with the text all lowercase
(run-experiment lower-case-transformer)
;; Run an experiment with the stop words removed
(run-experiment strip-stopwords-and-whitespace-transformer)
;; Run an experiment with the stop words removed and the text all lower case
(->> [lower-case-transformer strip-stopwords-and-whitespace-transformer]
     (compose-text-transformers)
     (run-experiment))
```

3.2 Tokenization

Tokenization is the process of breaking a large structure up into smaller parts that are easier to understand and process. In this project, two methods of tokenizing text are considered. The first method is to tokenize by individual characters. The second method is breaking the text up into individual words and symbols (like a hashtag, URL, or Twitter handle). Tokenization takes place after transformation of text. This makes it easier to apply the same methods of transforming text to the different methods of tokenization.

The simplest method of tokenizing text is by character. In Hy, this is a simple matter of calling the `list` function on some transformed text.

```
(defn tokenize-by-char [text &optional [transformer (fn [s] s)]]
  (->> text
       (transformer)
       (list)))
```

Tokenizing text by word can be more complicated, and for that, the NLTK library is used [1].

```
(defn tokenize-by-word [text &optional [transformer (fn [s] s)]]
  (import  [nltk.tokenize [word-tokenize]])
  (->> text
       (transformer)
       (word-tokenize)))
```

NLTK also provides a way to tokenize Tweets, which preserves hashtags, URLs, Twitter Handles, and text-based emoticons like ":-)" and "<3". It will also reduce

the amount of times a single character is used in a row. For example, reducing the Tweet "I looooooooooooove #MarkovChains!!!!!!!!!!!!!" to a more manageable and normalized "I looove #MarkovChains!!!" Here is the function for doing that.

```
(defn tokenize-tweet-by-word [text &optional [transformer (fn [s] s)]]
  (import [nltk.tokenize [TweetTokenizer]])
  (defn create-tokenizer []
    (TweetTokenizer :strip-handles False
                    :reduce-len True))
  (->> text
      (transformer)
      (.tokenizer (create-tokenizer))))
```

These tokenizing functions are used in conjunction with the transformers. One a transformer is obtained, either an individual or composed transformer, the text can be transformed and tokenized all at the same time using a tokenization function.

```
(setv transformers [lower-case-transformer
                    strip-stopwords-and-whitespace-transformer
                    strip-punctuation-transformer])
(->> transformers
     (compose-text-transformers)
     (tokenize-by-word text))
```

This evaluates to an ordered list of words that represent the text after it has been stripped of stop words, punctuation, and excess white space. The resulting list can then be used either to create a Markov chain or be analyzed with using a Markov chain that was created using the same transformation and tokenization methods.

3.3 Markov Chain Creation

The process of creating a Markov chain for a piece of text is the same for both characters and words. It is an iterative process that counts every state transition in the text, then calculates the probability of each transition occurring.

The algorithm is broken up into multiple parts. The first simply loops over the tokens and calls the `update-markov-chain` function on each iteration, providing the current state of the chain, and the next state. This represents a single transition in the process.

```
(defn create-markov-chain-single-text [txt markov-chain &optional (normalize True)]
  (for [i (range 0 (len txt))]
    (if (= i (dec (len txt)))
      None
      (update-markov-chain markov-chain (get txt i) (get txt (inc i)))))
  (if normalize
    (normalize-markov-chain markov-chain)
    markov-chain))
```

The `update-markov-chain` function takes in a Markov chain and two tokens. If the first token is not present in the chain (meaning this state has not been encountered yet), it adds that new state. Then, it will add the transition in if it doesn't exist, or increment the number of times that transition has happened. This builds up a count of all the transitions that have taken place so far in the Markov process.

```
(defn update-markov-chain [markov-chain token next-token]
  (if (None? next-token)
```

```
markov-chain
(do (unless (in token markov-chain)
      (assoc markov-chain token {}))
    (setv current-token (get markov-chain token))
    (if (in next-token current-token)
      (assoc current-token next-token (inc (get current-token next-token)))
      (assoc current-token next-token 1))
    markov-chain)))
```

Once the iterative process for building the Markov chain is done, a graph is returned that doesn't quite look like a Markov chain yet. Taking the example from the Introduction (section 1), here is what the chain would look like at this point.

```
{"p" -> {"r" 1},
 "r" -> {"e" 1, " " 2, "k" 1},
 "e" -> {"d" 1, " " 1, "r" 1, "s" 2},
 "d" -> {"i" 1},
 "i" -> {"c" 1, "n" 2, "t" 2, "s" 1},
 "c" -> {"t" 1, "h" 1},
 "t" -> {"i" 1, "h" 3, "w" 1, "t" 1, "e" 1},
 "n" -> {"g" 1, " " 1, "a" 1},
 "g" -> {" " 1, "e" 1},
 ...
 "y" -> {"s" 1}}
```

Once we have this graph, it is transformed into a Markov chain with probabilities instead of the number of transitions that occur in the text. This process of "nor-

malizing" the Markov chain is fairly simple. For each state, the total amount of transitions are counted, then each number of times the transition occurs is divided by the number of possible transitions for the current state. For example, in the state `"i"` in the Markov chain above, there are 6 total transitions. The probability of transitioning from state `"i"` to `"c"` is $\frac{1}{6}$, and the probability of transition from state `"i"` to `"n"` is $\frac{1}{3}$. Here is the code for normalizing a Markov chain.

```
(defn normalize-markov-chain [markov-chain]
  (for [word markov-chain]
    (setv word-dict (get markov-chain word))
    (setv total (reduce + (.values word-dict)))
    (for [k word-dict]
      (assoc word-dict k (/ (get word-dict k) total)))) markov-chain)
```

Once a Markov chain has been normalized, it can be used for analyzing text.

3.4 Text Analysis with Markov Chains

Two methods for predicting the author of text with Markov chains were considered. The first determines the probability that the text in question could be represented in a Markov Process described by a Markov chain for a known writer. The second method uses the existence of transitions in the chain to determine how much of the text could be represented in a Markov Process. This second method does not take into consideration the probability that a transition could occur, only the existence of a transition. The first method will be referred to as a probability analysis, and the second as a transition existence analysis.

3.4.1 Probability Analysis

One traditional use of Markov chains representing text is generating new text based on the transition probabilities present in a Markov chain. This process is done by first selecting a starting state. The next state is chosen by using a weighted random number generator. The weights used are the probabilities for the possible transitions for the starting state. Once the next state is randomly selected, the process repeats using the chosen state and it's transition probabilities to select the next state. This results in a random piece of text that looks similar to the original text used to create the Markov chain.

Using Markov chains for guessing the author of a piece of text is related to the process for generating text with a Markov chain. Essentially, the Markov chains are used to calculate the probability that the unknown text could be randomly generated by the Markov chain. Specifically, this process is calculating the probability that, starting in state 0 with a probability of 1, a Markov Process represented by the Markov chain will transition from state 0, to state 1, to state 2, ..., to state N, where N is the number of tokens in the unknown text.

Given the Product Rule for Conditional Probabilities, and the fact that the probabilities in a Markov chain are independent of one another (the probability of transitioning to one state does not depend on any of the previous states, just the one the Markov Process is currently in), the probability of going from state i to $i+1$ to $i+2$ is simply $P(i+1|i) * P(i+2|i+1)$ [6]. $P(i+1|i)$ is read as "the probability of $i+1$, given the current state i."

As an example, we can look at the Markov chain representing the title of this paper.

```
{"p" {"r" 0.5 "o" 0.5}
```

```
"r"  {"e" 0.25 " " 0.5 "k" 0.25}
"e"  {"d" 0.2  " " 0.2 "r" 0.2 "s" 0.4}
"d"  {"i" 1.0}
"i"  {"c" 0.166666 "n" 0.333333 "t" 0.333333 "s" 0.166666}
"c"  {"t" 0.5 "h" 0.5}
"t"  {"i" 0.142857 "h" 0.428571 "w" 0.142857 "t" 0.142857 "e" 0.142857}
"n"  {"g" 0.333333 " " 0.333333 "a" 0.333333}
"g"  {" " 0.5 "e" 0.5}
;; ...
"y"  {"s" 1.0}}
```

Given this short excerpt of text, "in", we can calculate the probability that can be generated by this Markov chain. Starting with state 0 = "i", and state 1 = "n", the probability of transitioning from state 0 to state 1 is taken directly from the Markov chain to be 0.333333.

If the excerpt is longer, say "inge", the probabilities are multiplied together for each transition. Each transition is show in Table 1.

Table 1: Transition Probabilities

From	To	Probability
i	n	0.333333
n	g	0.333333
g	e	0.5

It follows that the probability that a Markov Process represented by this Markov chain would transition from "i" to "n" to "g" to "e" is $0.333333 * 0.333333 * 0.5 = 0.055555$, or a 5.5 % probability.

As the number of transitions present in a Markov chain grow as well as the number of transitions to check for grow, the probabilities calculated quickly approach zero. In

the experiments performed in this project, one piece of text is checked against multiple Markov chains. The resulting probabilities are frequently extremely small. To make the results easier to understand, the magnitude of the probabilities is considered. This is done by taking the logarithm (base 10) of every probability then taking the largest non-zero probability and dividing all the logarithms of the probabilities by the largest one. Some examples are given in the following table.

Table 2: Adjusting Probabilities for Magnitude

p	$(log\ p)$	$(log\ p) \div (max\ (log\ p))$
$9.313225746154785 * 10^{-10}$	-20.79441541679836	0.7333333333333334
$3.725290298461914 * 10^{-09}$	-19.408121055678468	0.7857142857142858
$1.4901161193847656 * 10^{-08}$	-18.021826694558577	0.8461538461538463
$5.960464477539063 * 10^{-08}$	-16.635532333438686	0.9166666666666667
$2.384185791015625 * 10^{-07}$	-15.249237972318797	1.0

In the end, the highest result is always 1.0 unless all the original probabilities are 0.0. The others are scaled relative to the highest probability.

3.4.2 Transition Existence Analysis

In addition to calculating the probability that text can be represented by a Markov chain, the amount of text that can be represented in a Markov chain was also calculated and compared with the probability analysis.

This process is simple. It takes a tokenized text excerpt, and returns the number of transitions represented in the Markov chain divided by the overall number of transitions in the tokenized text excerpt. Here is the function that performs this calculation, given a Markov chain and the tokenized text.

```
(defn get-percent-text-is-represented [markov-chain tokenized-text]
  (defn transition-exists [t1 t2]
```

```
    (or (= t2 None)
        (and (in t1 markov-chain) (in t2 (get markov-chain t1)))))
  (loop [[tokens tokenized-text] [matched-transitions 0]]
        (if (empty? tokens)
            (/ matched-transitions (len tokenized-text))
            (recur (list (rest tokens))
                   (if (transition-exists (first tokens) (second tokens))
                       (inc matched-transitions)
                       matched-transitions)))))
```

While this does not take into account the probability of any transition occurring in the Markov chain, it requires no extra information, and can be substituted in place of the function that calculates the probability that the text is represented in a Markov chain.

Looking back at the example of the title of this paper, "Predicting the Author of Twitter Posts with Markov Chain Analysis", the following example calculates the percentage of the excerpt "Twitter Posts with Markov Chain Analysis" that can be represented in the Markov chain created with character tokens.

```
(->> (list "Twitter Posts with Markov Chain Analysis")
     (get-percent-text-is-represented title-markov-chain))
```

Which gives the result 1, meaning 100% of the excerpt can be represented in the title. A more interesting excerpt would be "Twitter is alright," which does not appear in the title that the Markov chain is created from.

```
(get-percent-text-is-represented title-markov-chain (list "Twitter is alright"))
```

This returns 0.61111, meaning 61% of that excerpt can be represented in the title. Note the word "is" is not present in the title, but the transition from "i" to "s" is present in the title, meaning it can be represented in the title's Markov chain.

3.5 Project Gutenberg Control Experiments

Text from Project Gutenberg was used to perform control experiments before the experiments were run using text from Twitter.

Nine writers with multiple works available in the public domain were chosen. A select number of their works were used to create Markov chains, and each work was represented in its own Markov chain. Each text was shortened to the first $3000 * 140 = 420,000$ characters. This number is chosen to reflect the length of the Twitter corpora created for a single user account, where the number of Tweets taken from every account is between 2000 and 3000, with each Tweet being 140 characters long.

For the works of every author used this way, a random excerpt of a specified length from the text was taken. The likelihood that each excerpt can be represented in each Markov process in the collection, excluding the text from which the excerpt was taken, was recorded. The Markov chain producing the highest probability of generating the excerpt was used to guess the author. Because the original work from which the excerpt was taken was already known, the correctness of the guess was also known. Once this guessing process was completed for each work for each author, the percentage of correct guesses was computed and recorded.

This was done for multiple control groups, each group having text that is transformed in a different way. For example, one group may only have excess white space removed, while another may have all the punctuation stripped out. All the control groups had

experiments run with different excerpt lengths. For each excerpt length and control group, a set of five experiments were run, each using different random excerpts of the same length, yielding the percentage of correct guesses. The average of all five experiments were used to determine the accuracy of the control group's tokenization and transformation strategy at the specified excerpt length.

Part of the code for running the experiments is given here. The main function takes in a data structure storing an author, the title of each work, and the Markov chain for that work. It also accepts the minimum starting index from which the excerpt is taken, and the length of the excerpt. A function named `probability-fn` is passed in to specify which type of analysis will be done, which will be either a probability analysis or a transition existence as described in section 3.4. It will return a data structure that stores the author, the title of each work used, the random excerpt that was taken, a list of the probabilities calculated by the `probability-fn`, and the author guessed to have written it.

```
(defn run-experiment
  [experiment-num excerpt-start excerpt-length probability-fn tokenizer]
  ;; probability-fn takes in a Markov chain and
  ;; a list of Tokenized text, and returns a probability
  (defn take-random-excerpt [book-text]
    (->> (-> (randint excerpt-start (- (len book-text) excerpt-length))
             (drop book-text))
         (take excerpt-length)
         (list)))
  (defn create-probability-mapper [excerpt]
    (fn [current-book]
```

```
     {:author (get current-book :author)
      :title (get current-book :title)
      :probability (probability-fn (:markov-chain current-book) excerpt)}))

(defn get-best-guess [probability-results]
  (setv best (->> probability-results
                  (map (fn [res] (get res :probability)))
                  (list)
                  (max)))
      (->> (filter (fn [x] (= (get x :probability) best)) probability-results)
           (list)
           (first)))
;; Get the transformer to use
(setv transformer (-get-transformer experiment-num))
;; Create the markov chains for each book
(setv markov-chains
  (-create-control-markov-chains authors transformer tokenizer))
(defn experiment-mapper [book]
  ;; Grabs a random excerpt from book,
  ;; then maps the excerpt to the probability analysis results
  (setv book-text (->> (:text book)
                       (transformer)
                       (tokenizer)))
  (setv random-excerpt (take-random-excerpt book-text))
  ;; The book the current excerpt is taken from is removed
  ;; before the probabilities are calculated.
```

```
    (setv probabilities (->> markov-chains
                             (filter (fn [a] (!= (:title book) (:title a))))
                             (map (create-probability-mapper random-excerpt))
                             (list)
                             (adjust-result-probabilities
                               (= probability-fn get-probability))))
    (setv best-guess (get-best-guess probabilities))
    ;; False if wrong author, guess is 0 probability, or all probabilities are 1
    (setv correct? (and (= (:author book) (:author best-guess))
                        (not (zero? (:probability best-guess)))
                        (not (->> (map (fn [r] (:probability r)) probabilities)
                                  (map one?)
                                  (reduce and)))))
    {:author (:author book)
     :title (:title book)
     :excerpt random-excerpt
     :probabilities probabilities
     :best-guess best-guess
     :correct? correct?})
  ;; This kicks off the experiments
  (list (map experiment-mapper markov-chains)))
```

Here is an example of a single entry in the data structure used to analyze the results using character tokenization.

```
{:author "Melville, Herman"
 :title "Typee: A Romance of the South Seas"
```

```
 :excerpt "cloth, trimmed with yellow silk, which, descending a
           little below the knees, exposed to view he"
 :probabilities [{:author "Melville, Herman"
                  :title "Battle-Pieces and Aspects of the War"
                  :probability 0.9759814377553008}
                 {:author "Melville, Herman"
                  :title "The Piazza Tales"
                  :probability 1.0}
                 {:author "Austen, Jane"
                  :title "Sense and Sensibility"
                  :probability 0.9855037010312496}
                 ;; ...
                 {:author "Shakespeare, William"
                  :title "The Taming of the Shrew"
                  :probability 0.959312781543868}]
 :best-guess {:author "Melville, Herman"
              :title "The Piazza Tales"
              :probability 1.0}
 :correct? True}
```

This data structure is used to determine if the correct author was chosen as the best guess, as well as to examine the other probabilities calculated by the Markov chains. By the end of all the analysis all the entries in the data structure are reduced to a single number representing how many excerpts were correctly matched with their author by this function.

```
(defn -calc-correct-guess-percentage [experiment-results]
```

```
(->> experiment-results
     (map (fn [r] (:correct? r)))
     (filter (fn [v] v))
     (list)
     (len)
     ((fn [l] (/ l (len experiment-results))))))
```

For each experiment run, the final result is the number of correct guesses divided by the total number of guesses, giving a simple accuracy rating.

$$\text{Accuracy } = \frac{\text{Number of correct guesses}}{\text{Total number of guesses}} \tag{1}$$

The experiments were run a number of times, and the average of those final accuracies gave the average accuracy for a single control group.

3.6 Twitter Experiments

The Markov chain analysis was done on forty different Twitter accounts. Similar to the control groups there were be multiple Twitter groups, each with text transformed in different ways. For each account, their corpus of Tweets was represented by a single Markov chain. For every account, a Tweet was removed from the corpus, and the probability that each account could generate the Tweet was computed. Note that the removed Tweet was not included in the Markov chain representing the corpus from which the Tweet was taken. The account with the highest probability of generate the Tweet was guessed to be the creator of the removed Tweet. The original author of the Tweet was known, therefore the correctness of that guess was also known. This was be done for each Twitter account, and the percentage of correct guesses

was computed and stored. This defines one experiment, and was done for each group multiple times. Once the results from all the experiments were collected, the average result for each group was taken to determine the most accurate group, which also gave the most accurate method for tokenization and transformation.

Unlike the control experiments, it was not necessary to take random excerpts of the text under question, since the length of individual Tweets has an upper bound of 140 characters (all the data used was gathered before Twitter extended the maximum Tweet length to 280 characters). The focus for the Twitter analysis is on running the experiments with different combinations of tokenization and transformations, not excerpt length.

The following is the code for running a single round of experiments. It takes a random Tweet from every user, removes that Tweet from the user's collection of Tweets, then calculates the probability the Tweet was created by a Twitter handle, for every Twitter handle in the collection. Taking the Twitter handle with the highest probability for creating a Tweet, it chooses that to be the predicted creator of the Tweet.

```
(defn -run-experiment [tweet-corpus group-num tokenizer probability-fn]
  (setv transformer (-get-transformer group-num))

  (defn handle-mapper [handle]
    "Takes a twitter handle and creates a data structure containing the handle,
     a random tweet, a collection of all the handle's tweets, and a Markov chain
     representing all the tweets."
    (setv tweets (get tweet-corpus handle))
    (setv random-tweet (->> (-take-random-tweet tweets)
```

```
                          (transformer)
                          (tokenizer)))
  (setv tweets (->> tweets
                    (remove (fn [t] (= t random-tweet)))
                    (list)))

  ;; The returned data structure
  {:handle handle
   :random-tweet random-tweet
   :tweets tweets
   :markov-chain (-tweets-to-markov-chain tweets transformer tokenizer)})

;; Creates a complete list of the data structures created by handle-mapper
(setv twitter-markov-chain-map (->> tweet-corpus
                                    (map handle-mapper)
                                    (list)))

(defn experiment-mapper [handle-map]
  "Maps elements in twitter-markov-chain-map to a data structure similar
   to the one created by handle mapper, but with the experiment results
   merged in. They are
   :probabilities [{:handle handle1 :probability p1}
                   {:handle handle2 :probability p2}]
   :best-guess {:handle n}
   :correct? True/False"
```

```
(defn probability-mapper [e]
  "Takes an entry from the twitter-markov-chain-map, and maps the probability
   that each chain could generate the random tweet from handle-map"
  {:handle (:handle e)
   :probability (probability-fn (:markov-chain e) (:random-tweet handle-map))})

;; Calculate all the probabilities that a tweet could be produced from each
;; Markov chain in the corpus
(setv probabilities (->> (map probability-mapper twitter-markov-chain-map)
                         (list)
                         (adjust-result-probabilities
                           (= probability-fn get-probability))))

(defn best-guesser [p1 p2]
  (if (>= (:probability p1) (:probability p2))
      p1
      p2))

;; Find the handle with the greatest probability of generating the random tweet.
(setv best-guess (->> probabilities
                      (reduce best-guesser)))

;; Merge the results of the experiments with the handle map
(merge handle-map {:probabilities probabilities
                   :best-guess best-guess
                   :correct? (and (= (:handle best-guess) (:handle handle-map))
```

```
                                        (not (zero? (:probability best-guess)))))}))

  ;; Kick off the experiment
  (->> twitter-markov-chain-map
       (map experiment-mapper)
       (list)))
```

Shown here is a single entry of the data structure returned when trying to identify who wrote this historic tweet written by realDonaldTrump.

```
{:handle "realDonaldTrump"
 :random-tweet "Despite the constant negative press covfefe"
 :tweets       ;; tweet collection, to large to show
 :markov-chain ;; Markov chain, to large to show
 :probabilities [{:handle "realDonaldTrump" :probability 0.9690505095437781}
                 {:handle "arthur_affect" :probability 0.9908391563211951}
                 ;; ...
                 {:handle "SethAbramson" :probability 1.0}]
 :best-guess {:handle "SethAbramson" :probability 1.0}
 :correct? False}
```

Similar to the control experiments, the accuracy was calculated by dividing the number of correct guesses by the overall number of guesses. The experiments were run a number of times and the average of all those accuracies was used to give the final accuracy for a single control group.

4 Results

4.1 Project Gutenberg Control Experiments

The control experiments are performed using data from nine authors with works in the Public Domain. Each author has about three or four works chosen, with twenty nine works in total. All of the texts are obtained from Project Gutenberg [7]. The authors and their works are listed in table 3.

For all of the control experiments, seven different control groups were used. The groups are defined by the different text transformations used on the books. All groups used the same set of authors and books. The text was tokenized by both characters and words for every control group. Table 4 describes each type of text transformation that is done on the control groups 1-7. What is not depicted in the table is that every control group has all excess white space normalized.

4.1.1 Probability Analysis By Character

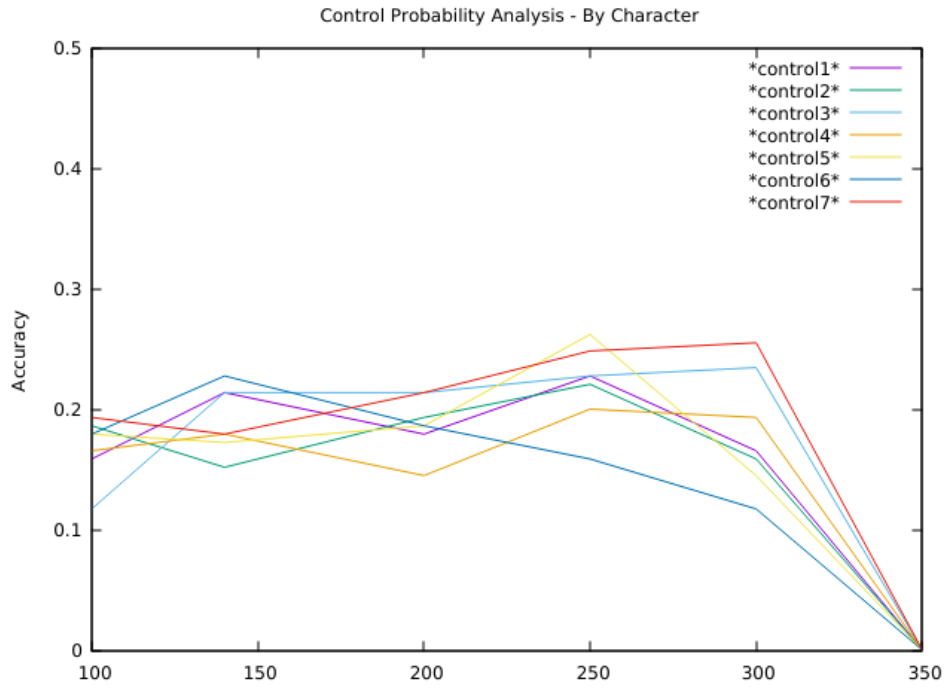
Performing a probability analysis with character tokenization is one of the more effective ways to look at books. The most accurate result is a 26% prediction rate with control group 7 using 300 character excerpts. Once the length of the excerpts reaches around 350 characters, the probabilities produced by the Markov chains are 0%. This is due to the probabilities tending towards zero and eventually becoming smaller than machine epsilon, which for the double precision data types used in this project is roughly 2^{-53}.

Table 3: Authors and Books Used in Control Experiments

Author	**Book**
Austen, Jane	Sense and Sensibility Emma Pride and Prejudice Persuasion
Carroll, Lewis	Alice's Adventures in Wonderland Through the Looking-Glass The Hunting of the Snark: An Agony in Eight Fits
Dickens, Charles	Oliver Twist A Christmas Carol The Seven Poor Travellers
Doyle, Arthur Conan	The Adventures of Sherlock Holmes The Tragedy of the Korosko The Firm of Girdlestone The New Revelation
Irving, Washington	The Sketch-Book of Geoffrey Crayon Tales of a Traveller Bracebridge Hall, or The Humorists
Melville, Herman	Typee: A Romance of the South Seas Battle-Pieces and Aspects of the War The Piazza Tales
Shakespeare, William	Hamlet, Prince of Denmark A Midsummer Night's Dream The Taming of the Shrew
Twain, Mark	The Prince and the Pauper A Tramp Abroad A Horse's Tale
Wilde, Oscar	The Soul of Man under Socialism The Picture of Dorian Gray Charmides, and Other Poems

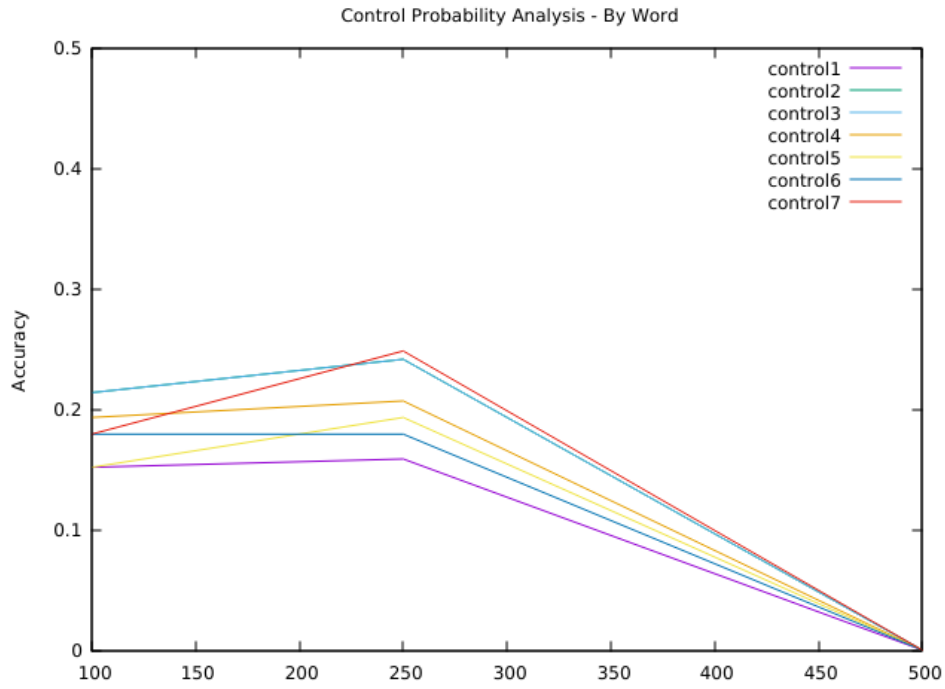
Table 4: Control Group Descriptions

Group	**Punctuation Removed**	**All Lower Case**	**Stop Words Removed**
control 1			
control 2		X	
control 3	X	X	
control 4	X		
control 5	X		X
control 6		X	X
control 7	X	X	X



4.1.2 Probability Analysis - By Word

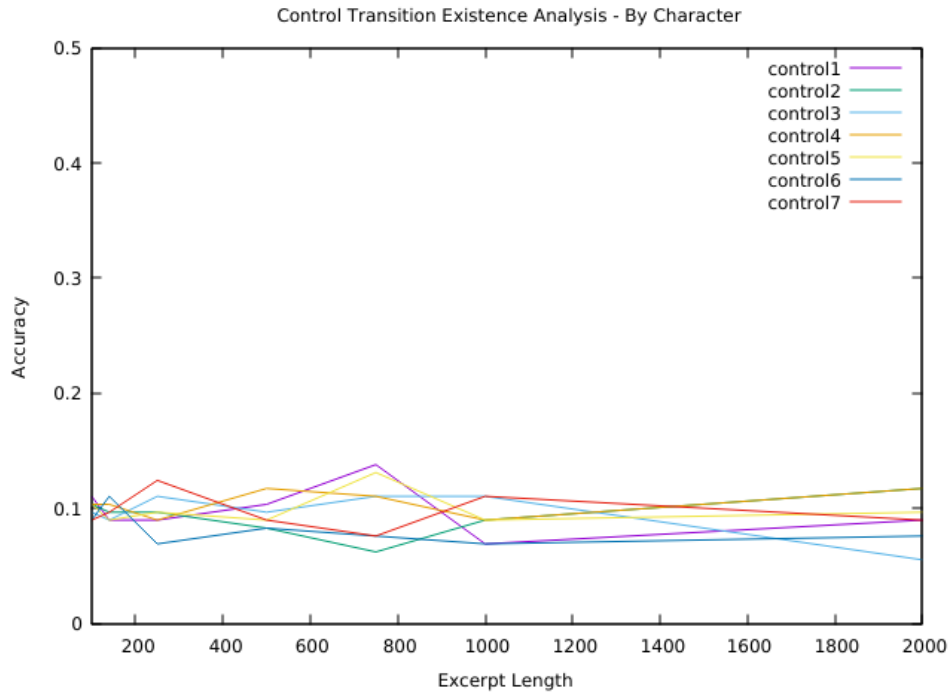
Performing a probability analysis with word tokenization was extremely ineffective. Most often, this is because the transitions found between words are frequently close to zero, so small they cannot be represented in double precision floating point numbers and effectively zero, or simply non-existent. This causes the probability of just about every Markov chain to go to zero pretty quickly. When checking against the entire text of a book, the results were always zero, but choosing to limit the length of the texts to 420,000 characters used to create the Markov chains resulted in a few books being guessed correctly.



4.1.3 Transition Existence Analysis - By Character

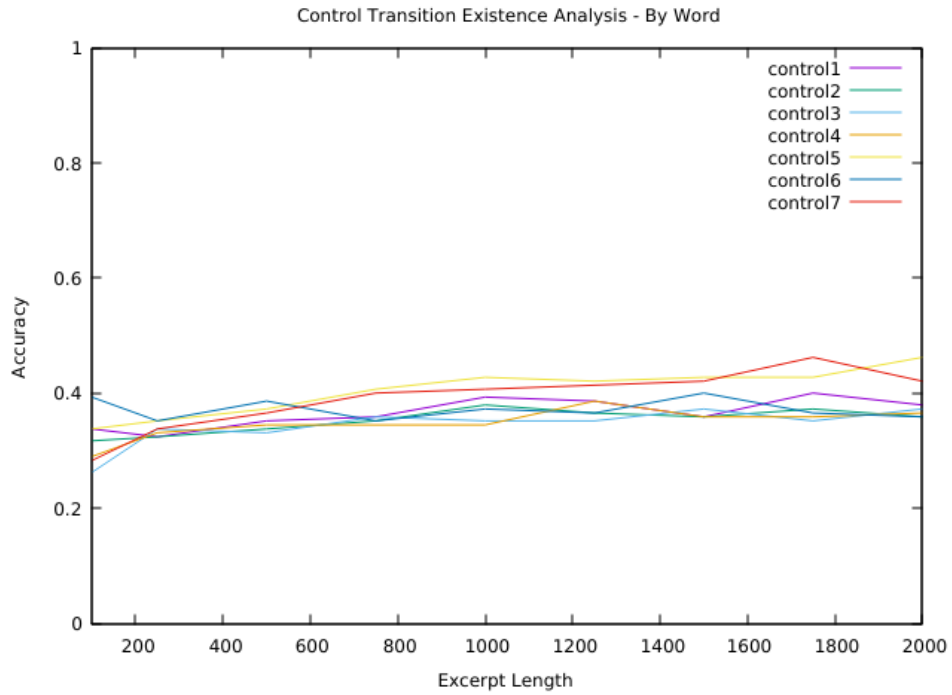
For the experiments done with the transition existence analysis, the same control groups were used as with the probability analysis.

Looking at the transition existence analysis, when done by character an accuracy of about 10% is all that can be achieved. These results are due to some authors who occasionally use uncommon characters like accented a's, or uncommon punctuation. Other than that, considering all texts are written in English, the number of transitions from a string of English text existing in any Markov chain will be very close to, if not exactly 100%. As a result, the analysis has trouble picking the author out when all the results are at or near 100%. Usually the author chosen will be one of the first ones in the list, meaning the correct guesses are usually just random chance and can effectively be ignored.



4.1.4 Transition Existence Analysis - By Word

The most accurate analysis done on the control texts the transition existence analysis when tokenizing all text by word. Being about between 30% and 40% accurate, the results are pretty consistent as the number of words in each random excerpt that is taken increases.



4.2 Twitter Experiments

The Twitter accounts used in the experiments are described in Table 5. All the Twitter handles used are taken from popular journalists and news personalities from various main stream media outlets. All of the accounts report on similar topics which should, in theory, make the analysis depend more on their individual writing style and less on the unique proper nouns, hashtags, and handles used in their Tweets.

Since only one text length of 140 characters was used for each Tweet as opposed to the various excerpts used in the control experiments, each individual experiment is run more times than before. For all the Twitter experiments, eight different groups were used. The text transformation defining each group is described in Table 6. It is not shown in the table that each group has excess white space removed. There were also no experiments run with punctuation removed in order to preserve the

Table 5: Twitter Handles

@andersoncooper	@ggreenwald
@maddow	@joenbc
@jaketapper	@daveweigel
@gstephanopoulos	@markknoller
@camanpour	@mitchellreports
@ezraklein	@michellemalkin
@natesilver538	@mharrisperry
@mikeallen	@howardkurtz
@chucktodd	@danaperino
@brianstelter	@bretbaier
@buzzfeedben	@greta
@ariannahuff	@ahmalcolm
@chrislhayes	@glennbeck
@anncurry	@donnabrazile
@nytimeskrugman	@davidcorndc
@megynkelly	@SethAbramson
@seanhannity	@adamgoldmanNYT
@wolfblitzer	@maggieNYT
@thefix	@PekingMike
@fareedzakaria	@arthur_affect

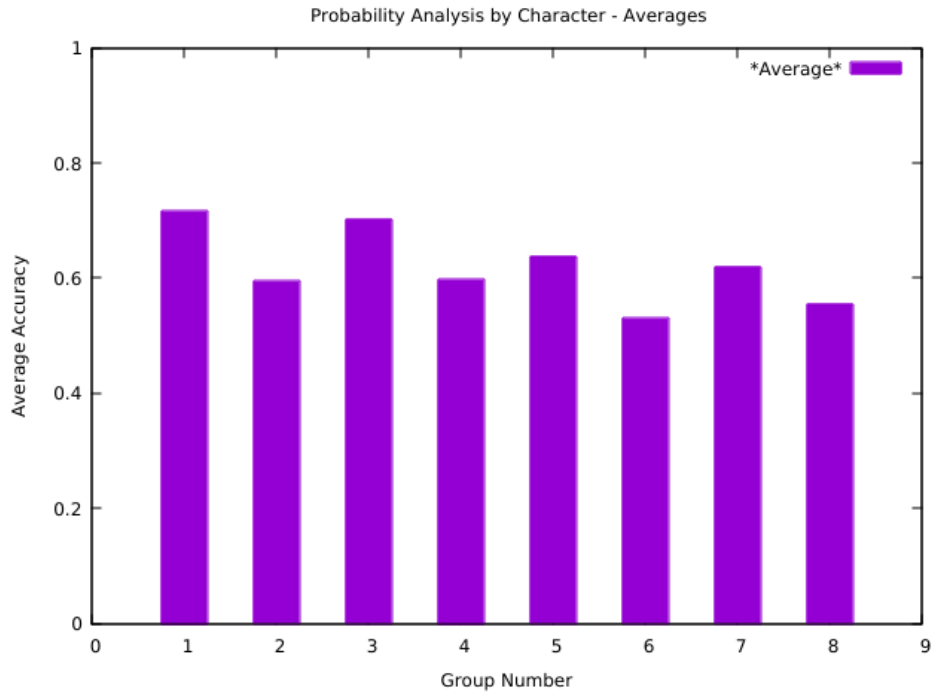
emoticons, URLs, and hashtags found in many Tweets.

Table 6: Twitter Analysis Group Descriptions

Group	**Lower Case**	**No Words Removed**	**URLs Normalized**	**Handles Normalized**
group 1				
group 2	X			
group 3		X		
group 4	X	X		
group 5			X	X
group 6	X		X	X
group 7		X	X	X
group 8	X	X	X	X

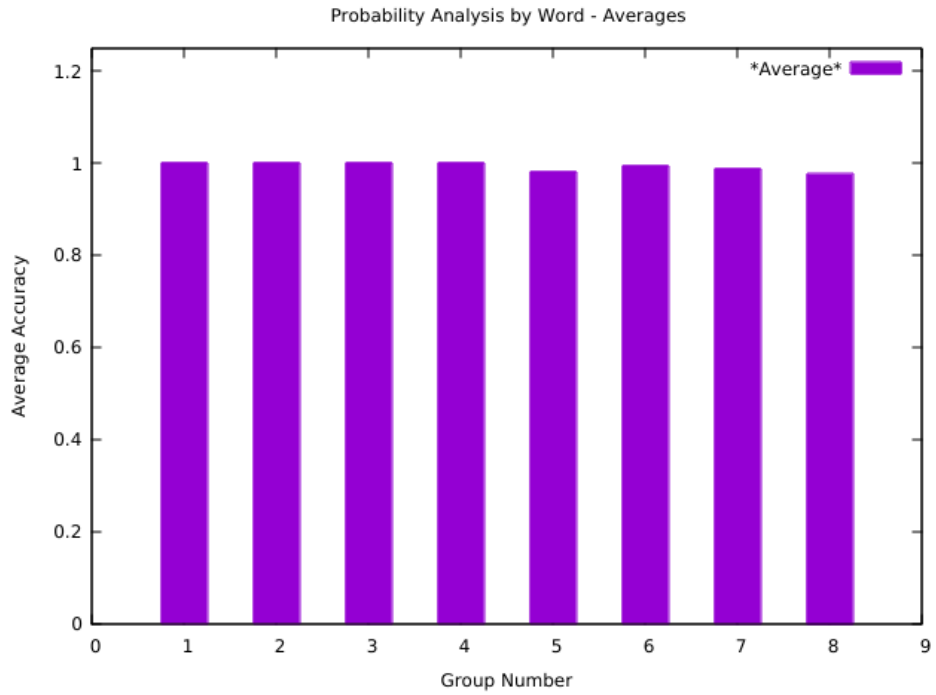
4.2.1 Probability Analysis - By Character

The results from this probability analysis are very promising. Compared to the analysis done on the control experiments, this is about 50% more accurate. The best results come from not doing any text transformations besides removing extra white space. One thing to note is that converting everything to lower case text is always less accurate than the matching control group that does not change the case of any of the text.



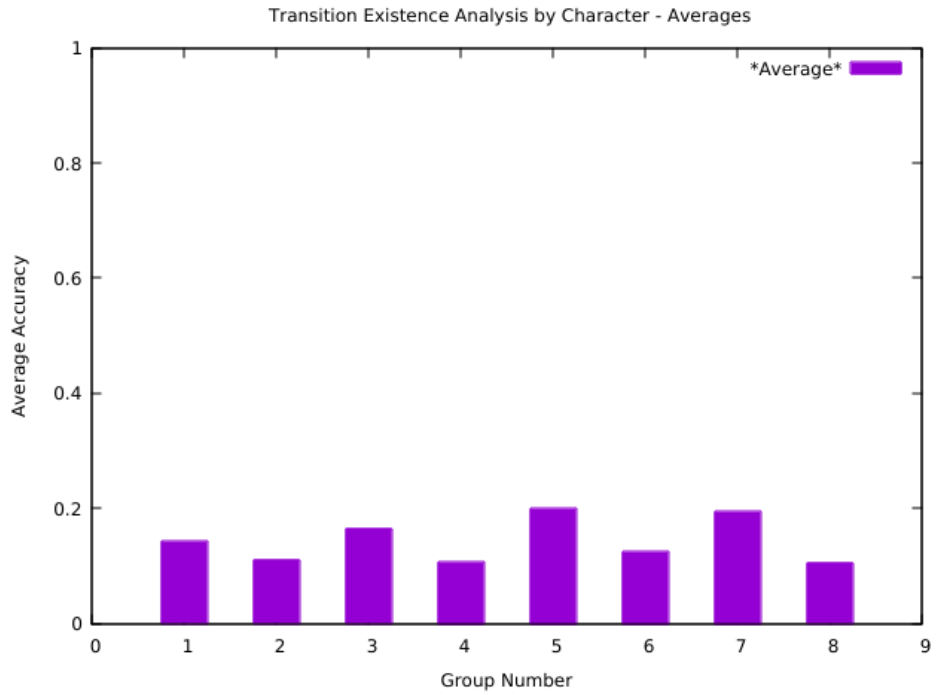
4.2.2 Probability Analysis - By Word

Doing the probability analysis by word is always close to, if not at 100% accuracy. The explanation for this is due to words being used by one Twitter user not occurring in another user's Tweets as often, or at all. This results is being able to predict the author of a Tweet very accurately. These results might start to drop if the Twitter accounts used in the comparison frequently Tweet about the same subject, participate in the same reply threads, or used common hashtags.



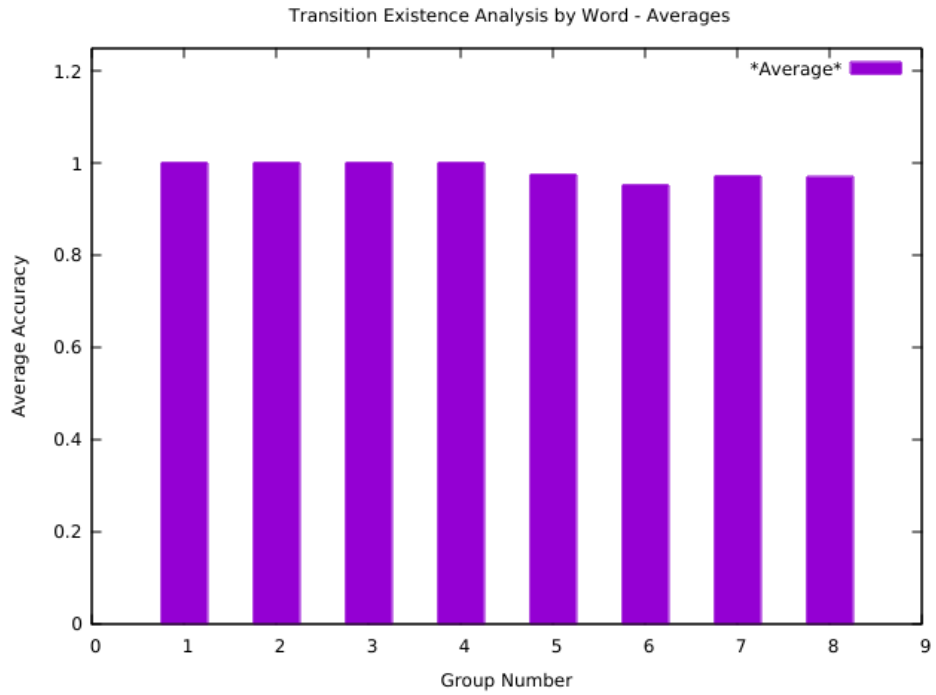
4.2.3 Transition Existence Analysis - By Char

Similar to the control experiments, the same text transformation groups used for the probability analysis are used in the transition analysis. Also similar to the control experiments, this method is not very effective, and for the same reason. One thing to note is that accuracy is consistently lower when the text is transformed to all lower case.



4.2.4 Transition Existence Analysis - By Word

For the same reason that the probability analysis is extremely effective when tokenized by words, the transition existence analysis is also very effective. The use of uniquely misspelled words, hashtags, and mentions of other users contributes to this. It does not seem to matter which methods for processing the text are used, all produce about the same accuracy.



5 Conclusions

5.1 Hy

This entire project could have been written in Python just as easily as it was written in Hy, if more easily. Hy as a language is immature, and went through two major releases with breaking changes during the course of this project. Since the language is young, the tooling for it is also lacking. In fact, some tooling was written and updated during this project by the author to make working with Hy easier [8] [9].

That being said, having access to the entire Python ecosystem from a dialect of LISP had many advantages. Being able to interact with a read-eval-print-loop (REPL) allowed for very rapid feedback while developing the code base. The use of the

Hy REPL allowed for functions to be redefined and executed without having to re evaluate the rest of the code. Using macros available in Hy also simplified some of the code that would be more troublesome to write and update in Python.

Since Python is already widely used for various types of scientific research, the resulting Python code produced by the Hy code written in this project was very efficient. Some tools built into Hy do not leverage this feature of Python very well. In particular, the tail-call recursion features in Hy do not work well with the built in Python tools for list processing. Python's list processing features are built around using `for` loops, not recursive function calls. Some places in the code used the `loop/recur` macro, which implements tail call recursion in Hy, in places where `for` loops were more effective. In particular, the use of `loop/recur` made building Markov chains run in `O(n!)` time. Building Markov chains for project Gutenberg texts would take about six hours on a modern computer. Converting these recursive calls to `for` loops reduced that time to a few minutes, running in `O(n)` time. Thankfully, as seen in some of the code examples, `for` loops are trivial to implement in Hy.

If Hy is able to mature more and the community developing the language creates more user friendly tools, it would be a fine choice for writing code for research projects like this one in the future.

5.2 Control Experiments

The results of the control experiments show that the probability analysis is not very effective when trying to identify the authors of books. When compared against Twitter posts, books are much more formal. They do not contain many foreign characters, abbreviations, or any emojis that make a writer's style easy to identify.

As a result, the Markov chains become similar, especially with larger books. The use of common stop words (e.g. "the", "as", and "it") most likely contributes to this. The results seem to reflect this since the control groups that removed all the stop words (groups 5, 6, and 7) gave the best results compared to the control groups that kept the stop words in. Removing them allows for more proper nouns, frequently used adjectives, and verbs to define the most common transitions in a Markov chain.

Overall, the best results come from using the transition existence analysis when tokenizing excerpts by word and not character. The transition existence analysis also produces results consistently for excerpts of different lengths, whereas the probability analysis drops off after 350 characters or 250 words. The most effective way to pre-process the text before working with it are the transformations outlined in control group 7. The stop words are removed, the punctuation is removed, all excess white space is removed, and all the capital letters are converted to lower case.

5.3 Twitter Experiments

The results of the Twitter experiments show that it is easier to identify people who post on Twitter than authors of books in the public domain. Since Twitter posts are not held up to the same standards that published books are, there are more unique transitions from character to character and word to word. Twitter is known for pioneering the use of hashtags, which encourages users to combine phrases into a single "word". This, along with the ability to mention another user with the "@" character, creates more unique transitions that make people easier to identify. The recommend method for trying to identify users is the probability analysis combined with word tokenization. The transformation method that is most effective is to only remove excess white space. This methodology is able to predict the author of a Tweet

with 99% accuracy from the small set of 40 accounts considered in this project.

5.4 Final Thoughts

Overall, it can be said that the methods used in this project to identify writers does not work very well for long books. Other studies, such as those by Khmelev and Stanko have been shown to be more effective at identifying the authors of books [5] [3]. However, when it comes to identifying Twitter accounts, this method works well on a small scale.

Some interesting ways to expand this research would be to try and identify automated user accounts on Twitter, often called bots. In theory, posts by bots would be similar enough to one another that a set of texts from known bots could be used to identify potential new bot accounts, which are created on a daily basis. This combined with other indicators, such as time between posts, creation date, location, and number of followers could be an effective way to detect bot accounts used to spread false information or online scams.

References

[1] NLTK, "Nltk 3.2.5 documentation." `http://www.nltk.org/`, Sep 2017. Accessed: 2017-11-09.

[2] HyLang, "Welcome to hy's documentation!." `http://docs.hylang.org/en/stable/`, Oct 2017. Accessed: 2017-11-09.

[3] S. Stanko, D. Lu, and I. Hsu, "Whose book is it anyway? using machine learning to identify the author of unknown texts." `http://cs229.stanford.edu/proj2013/StankoLuHsu-AuthorIdentification.pdf`, 2013. Accessed: 2017-11-09.

[4] D. Tran and D. Sharma, "Markov models for written language identification." `https://www.semanticscholar.org/paper/Markov-Models-for-Written-Language-Identification-Tran-Sharma/2bf08addb83f51befa8b4bc7ed16b54ed34018d0`, 2005. Accessed: 2017-11-09.

[5] D. Khmelev, "Using markov chains for identification of writers," *Literary and Linguistic Computing*, vol. 16, p. 299–307, Jan 2001.

[6] R. B. Campbell, "Conditional probability and the product rule." `http://www.cs.uni.edu/~campbell/stat/prob4.html`. Accessed: 2017-11-09.

[7] P. Gutenberg, "Project gutenberg." `http://www.gutenberg.org/`, Aug 2017. Accessed: 2017-11-09.

[8] D. Freeman, "Pull request 6357 - syl20bnr/spacemacs." `https://github.com/syl20bnr/spacemacs/pull/6357`, Jun 2017. Accessed: 2017-11-09.

[9] D. Freeman, "Pull request 9033 - syl20bnr/spacemacs." `https://github.com/syl20bnr/spacemacs/pull/9033`, Jun 2017. Accessed: 2017-11-09.