university of groningen

Bachelor Thesis

Mathematics

Random Writing And Authorship

Author
Frank Lefeber

Supervisors
Prof. Dr. Ernst Wit
Dr. Javier G. Hernandez

July 11, 2013

Abstract

The question of authorship of texts has already been investigated by several scientists. For example, in the Journal Of The American Statistical Association, authorhip was determined with the help of context free words that were called function words. [6] In this paper a different method for determining the authorship of a text will be explored and analysed. The verdict of the origin of a text will be based on likelihood ratio tests between candidate authors. The loglikelihoods that are necessary are unknown, but they will be estimated. These estimates are derived from the probability that the candidates write the text using a stochastic model. This model is designed to simulate the writing style of an author. It contains a Markov Chain based on n-grams; transitions between n-tuples of words in the text. It will use the maximum likelihood estimator to assign probabilities to each transition. For some analysis, the topic of ergodicity will be briefly covered, along with the corresponding conditions. In order to test whether the method is suitable for authorship testing, we built the required functions in MATLAB®. There is also a section devoted to the way the method was implemented in this programming language.

1 Problems

The problems that are considered in this paper correspond to the goals mentioned above, some are immediate. The model that will simulate an author is a random text generator. Note that generating random text is not the same as randomly generating text (the model will do the latter). But what exactly will the model look like? How will it ensure grammatical correctness and in what degree? How will it simulate an author and can this be done with a random text generator? When we are able to answer these questions, more questions arise. How do we use our random text generator to test authorship of texts? What do we know about the candidate authors? What size of text works best for an authorship problem? How will we determine authorship? What treshhold will the likelihood ratio test use?

For each problem there are multiple approaches. Consider the question: "What method can be used for a random text generator?" It has multiple answers. There is the option of creating a string of 'words' from a set of characters by just randomly choosing characters and putting spaces after a certain amount of characters. This process can be randomised more by allowing an interval for the length of the 'words', but this will not prevent the creation of an insane amount of gibberish. Another model is one that uses a Markov Chain to choose letters, based on what letters are already present. What a Markov Chain is will be explained in the next section. This method could use a so called n-gram model that bases its choices on the n previously chosen letters, these choices are called transitions. Special cases of this model are the unigram, bigram and trigram models (for $n = 1, 2, 3$). We will be using these models ourselves, but not for letterbased transitions. The problem with a letterbased model is that you get nonexisting words for small n. It is also a bit inefficient to generate a whole book letter by letter. These thoughts make us think about wordbased transitions. A trigram model would for example choose a word based on the three words it follows. How this is done exactly is explained later.

Clearly, when the model was being developed, several choices were made. In this paper one can find the reasoning or theory behind these choices. The first topic that will be discussed is the theory of Markov Chains.

2 Markov Chains

A Markov Process is a random process that describes transitions of states that have the Markov Property, which means that each following state depends only on the current state. If such a process has a finite (or countable) discrete statespace, it is called a Markov Chain. [2] Markov Chains can be used to model a lot of problems that involve probability. A typical Markov Chain iteration looks like:

$$x_{t+1} = x_t P$$

An entry P_{ij} would correspond to the chance $Pr(X_{t+1} = j \mid X_t = i)$, which is the chance to land in state j when the current state is i. Note that it is possible to do multiple iterations at a time because it is logical that:

$$x_{t+2} = x_{t+1}P = (x_t P)P = x_t P^2$$

So it can be shown by induction that:

$$x_{t+n} = x_t P^n$$

So the chances of being in state j after n iterations are in column j of P^n. The entries P^n_{ij} correspond to those chances, given that the current state is i.

The next section provides some examples of Markov Chains, these should give us an idea of what they look like and how they can be created for random processes. After that section we can come up with a way to use them for the random text generator.

2.1 Examples Of Markov Chains

A very basic example is the random walk or drunken walk, where somebody can not walk straight and has certain chances of strafing left or right. The idea is that the person walks with a certain angle instead of going straight, so the entire walk would find place within a trianglular shape from the starting point.

This walk can be modelled with a Markov Chain, with equal chances of going left or right regardless of what happened during last step. To assure clarity, we put the new states on top of P and the current states to the left.

$$P1 = \begin{array}{c} \\ L \\ R \end{array} \begin{array}{c} \begin{array}{cc} L & R \end{array} \\ \begin{pmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{pmatrix} \end{array} \tag{1}$$

So the chance of taking one step to the left when the previous step was to the right is entry $P1_{21}$, which in this case is 0.5. If the pedestrian has some sense of compensation for imbalance, the chances of going left after going right and vice versa will be bigger. They could for example be 0.3 and 0.7:

$$P2 = \begin{array}{c} \\ L \\ R \end{array} \begin{array}{c} \begin{array}{cc} L & R \end{array} \\ \begin{pmatrix} 0.3 & 0.7 \\ 0.7 & 0.3 \end{pmatrix} \end{array} \tag{2}$$

Should the influence of the pedestrians concience have the opposite effect (inertia), the model could look like this:

$$P3 = \begin{array}{c} \\ L \\ R \end{array} \begin{array}{c} \begin{array}{cc} L & R \end{array} \\ \begin{pmatrix} 0.8 & 0.2 \\ 0.2 & 0.8 \end{pmatrix} \end{array} \tag{3}$$

Now the states are likely to repeat. Because the chances of going from one state to the other are the same for both states in the three previous models, the powers P^n of P converge to $P1$ from model 1. Also, the powers of $P1$ are not different from $P1$ itself.

But if the pedestrian would have a preference to stroll to the left in the last model, we get a different result for the powers of P. For

$$P4 = \begin{array}{c} \\ L \\ R \end{array} \begin{array}{c} \begin{array}{cc} L & R \end{array} \\ \begin{pmatrix} 0.95 & 0.05 \\ 0.2 & 0.8 \end{pmatrix} \end{array} \tag{4}$$

we get convergance to $\begin{pmatrix} 0.8 & 0.2 \\ 0.8 & 0.2 \end{pmatrix}$, which is different from $P1$. But we still see that each row of this limit matrix is the same. The reason behind this convergence of powers of P is explained in a following section.

A final model that could exist is one where the pedestrian starts falling to the left and can not control the situation. For this model, an example is:

$$P5 = \begin{array}{c} \\ L \\ R \end{array} \begin{array}{c} \begin{array}{cc} L & R \end{array} \\ \begin{pmatrix} 1 & 0 \\ 0.2 & 0.8 \end{pmatrix} \end{array} \tag{5}$$

Now, the state L is an absorbing state; once you enter, there is no way out. This is something we wish to avoid in a random text generator, as all randomness ends once the absorbing state is entered. To help understand Markov Chains better, there is a small section devoted to the visualisation of the process.

The process has a state sequence and a transition matrix. To help visualise the random walk, a graph containing all the information of the transition matrix can be made. In case of the first random walk, the directed graph, wherein arrows are transitions, would look like Figure 1.

The corresponding state sequence is part of a regular expression $(L \cup R)^*$ where * is a nonnegative integer, which is not fixed. It is "As often as you would like". Note that in the graph, any transitions from a state to itself needs no representation, as it equals one minus the sum of the transitions to other states. This is just for convenience, just like the absence of arrows with probability 0. Let us consider the graph in Figure 2 and find out what the regular expression for its state sequence is.

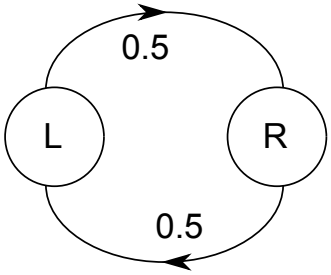

Figure 1: Graph For Random Walk

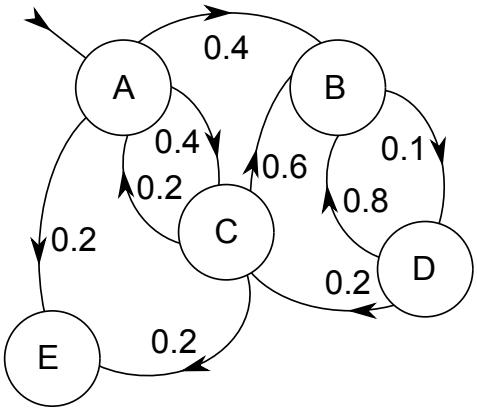

Figure 2: Graph For A Markov Chain

There is a total of five states A, B, C, D and E. Note that B is likely to repeat itself and there is an absorbing state E. The transition matrix P can be derived from the graph by just putting the values of each transition in the correct slot.

$$P = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{pmatrix} 0 & 0.4 & 0.4 & 0 & 0.2 \\ 0 & 0.9 & 0 & 0.1 & 0 \\ 0.2 & 0.6 & 0 & 0 & 0.2 \\ 0 & 0.8 & 0.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

The process starts at state A, from which there are options to go to B, C, E. From B there are transitions to B, D, from C there are options to go to A, B, E, from D there are transitions to B, C and state E is absorbing. So the regular expression is $A(CA)^*C^?(B^*(DB^*)^*DC((AC)^* \cup B^*(DB^*)^*DC)^*A^?E^*$, where $^?$ is 1 or 0.

Since E is an absorbing state, the last row of P^n will stay the same for any n. Because it is possible to go from any state to state E and E is absorbing, the limit of the powers of P is known.

$$\lim_{n\to\infty} P^n = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

For finite n though, there will always be nonzero chances that state E has not been reached yet. The largest of these chances will be in the second column, corresponding to state B. The convergence speed of P^n depends on the transitions to E and the transitions to the states that have transitions to E and so forth. We see that there are fairly large transitions to B. So convergence takes about a thousand iterations, with a tolerance level of 0.01. The largest eigenvalue of P is 1, this is always the case for Markov Chains, as will be shown in the next section. The second largest eigenvalue of P is 0.9953, which is 0.009 when raised to the power 1000. This is just below the tolerance level. The convergence speed seems to depend on the magnitude of the second largest eigenvalue. A transition from state I to J will be denoted by (I, J). If we change transitions $(D, C) \to 0.6$, $(D, B) \to 0.4$, $(C, E) \to 0.8$ and remove (C, B), there is convergence after about 80 iterations. The second largest eigenvalue of P now is 0.9424, which equals 0.0087 when raised to the power 80. We will not investigate this phenomenom further though, it is merely something interesting on the side.

The next section explains why the powers of the transitionmatrices converge to a certain limit and what this limit is.

2.2 Properties Of Markov Chains

An important notion in Markov Theory is irreducibility:

Definition 1. *A Markov Chain is called irreducible if there is a finite path bewteen any two states.*

Note that the model is derived from an input text, which is always finite. But a path between all states is not always granted. For example, there is no path from state E to any other state in the model of Figure 2.

Another important definition is that of periodicity:

Definition 2. *A Markov Chain is called periodic if all states are periodic.*

A state is called periodic if there is certainty for it to return to itself in qk steps, where k denotes the period and q is a set of integers in $\mathbb{N}$ larger than a certain q_0. The period k of state i is by definition the highest common factor of a set $\{n\}$ such that $P(X_n = i|X_0 = i) > 0$. Note that even if a state has period k, it may not be possible to return to itself in k steps. Example: If a state can only return to itself in $\{9, 12, 15, ...\}$ steps, its period is 3. But it can not return to itself in 3 steps. [2]

If $k = 1$ we call state i aperiodic. This means we can return to the state for any number of steps larger than q_0.

Theorem 1. *If any state in an irreducible Markov Chain is aperiodic, then all states are aperiodic.*

Proof. Suppose there is an aperiodic state. By definition of irreducibility there is a finite path from any state to the aperiodic state. There is also a finite path from the aperiodic state back to itself, a certain a steps with $a > a_0$ for some a_0. So suppose there is a periodic state with period k. From that state, there is a path of length qk for some set of $q's$ with $q > q_0$. There is a path to the aperiodic state of length b and there is a path of length c to reach the periodic state again. So all that remains to be done, is for us to do an appropriate a steps in between. That number, which depends on b and c, can be chosen such that $\nexists q$ for which $qk \,|\, a+b+c$ holds. Note that this is always possible, since $k \neq 1$. This yields a contradiction, so it can only lead to the conclusion that there can be no periodic state. □

Now that the definitions of irreducibility and periodicity have been given and observed, the definition of ergodictiy can be stated.

Definition 3. *A Markov Chain is called ergodic if it is aperiodic and irreducible. [8]*

The reason behind the convergence of the powers of the transitionmatrices is the following theorem:

Theorem 2. *If a Markov Chain with probability matrix P is ergodic, then there is a stationary probability measure $\pi = (\pi_1 ... \pi_N)$ such that $\pi = \pi P$. Furthermore, this π is a limiting distribution for the Markov Chain.*

Proof. An ergodic Markov Chain is irreducible and aperiodic, this has some immediate results. There is a path from any state i to itself for any number of steps higher than some q_0 due to aperiodicity. There is also a path to state i from any state of finite length because of irreducibility. So there is some finite number of steps q_i such that for any number of steps larger than q_i there is a path to state i from any other state. This means that column i of P^{q_i} has only positive entries. But this can be done for all states, so if we take $n = max\{q_1 ... q_N\}$, P^n is a positive matrix. So should P not be positive, P^n can be used. [7]

Now that P (or P^n) is a positive matrix, the Perron-Frobenius theorem can be used. This theorem states that P has an eigenvalue r that equals the spectral radius $\rho(P)$ and the modulus of every other eigenvalue of P is strictly smaller than r. [C] Gelfand's theorem states that the spectral radius equals the limit of $k \to \infty$ of $\|P^k\|^{\frac{1}{k}}$ for any matrix norm. [9] So an observation can be made for the 1-norm, using that P is rowstochastic. The definition of matrix norm is: [5]

$$\|P\|_1 = max\{\frac{\|Pv\|_1}{\|v\|_1} \mid \|v\|_1 = 1\}$$

The vector x is always stochastic, so its 1-norm is 1. For stochastic P the 1-norm will be one, because: [10]

$$\begin{aligned} \|Px\|_1 &= \sum_i \sum_j P_{ij} x_j \\ &= \sum_j \sum_i P_{ij} x_j \\ &= \sum_j x_j \sum_i P_{ij} \\ &= \sum_j x_j = \|x\|_1 = 1 \end{aligned}$$

So $\|P\| = 1$, which leads to $r = 1$. The Perron projection states that

$$\lim_{k\to\infty} \frac{P^k}{r^k} = vw^T$$

for v, w normalized right- and left eigenvectors respectively corresponding to r. [4] But $r = 1$, so this means that:

$$\lim_{k\to\infty} P^k = vw^T$$

It can be seen that $v = (1, ..., 1)$ is the right eigenvector, corresponding to 1. Since all rows of P sum to 1, it is obvious that $v = Pv$. The left eigenvector w corresponding to 1 is nothing but the solution of $(P - I)w = 0$, so let us call w^T π and we have $\pi = \pi P$. Now, since this v is just $(1, ..., 1)$, the limiting distribution is:

$$\lim_{k\to\infty} P^k = \begin{pmatrix} \pi \\ ... \\ \pi \end{pmatrix}$$

The 1-norm of these P^k is always 1, so $\pi_1 ... \pi_N$ must sum to 1. Besides that, probabilities are never negative, so the sum of the absolute values is also 1. So π is a probability measure. The left eigenvector π is called stationary, because if we view $x_{t+1} = x_t P$ as a dynamical system, π would be a stationary point. □

According to this theorem, if the transitionmatrices that are derived from input texts are ergodic, they have stationary probability measure π. So, when an author is simulated by a transitionmatrix, a claim can be made that the author has a certain distribution of states. So the index with the highest value of π is the authors most favourite state. This state can be a word, -pair or -triple, based on what model is used. This means that perhaps there is another possible method to test authorship. A set of the most favourite states could be chosen and compared per author. But this will not be discussed in this paper.

3 Model Ideas

The model will be a random text generator that is supposed to mimic the writing style of an author. Therefore, it needs to depend on a text by an author. This text will be called the **input text**. Besides that, the generated text should be reasonably legible, so it needs some grammatical structure. The most basic model would randomly choose words that are proportionally uniformly distributed. Later models will be more sophisticated.

3.1 The Most Basic Model

A reasonable chance for a word to be picked is just its number of occurrences divided by the total number of words in the input text. This is a unoform distribution, for which the chances of identical states will be summed, making it proportionally uniform. In the transitionmatrix of a model like this, each row will be identical. So if ν would be the vector that assigns chances to the words in vector x, the system would look like this:

$$x_{t+1} = x_t P = x_t \begin{pmatrix} \nu \\ ... \\ \nu \end{pmatrix} \tag{6}$$

This model would generate very random texts, in fact it will likely be too random. Consider this model for a small input text:

Mark draws a picture on a banana.

This sentence has 7 words, so the chance for any word at any step is $\frac{1}{7}$. Note that the word a occurs twice, so it has probability $\frac{2}{7}$. Let us think about the generated text before we generate anything. If a legible text is desired, the model will have to be able to end sentences with periods. It could randomly add those after certain numbers of generated words, but that would be a bad idea for a sophisticated text generator. So let us consider the word "*banana*" and the period that follows a single state "*banana.*" and use that instead. The same can be done with commas and other punctuation.

Now consider some examples of possible output. The sentence: "Mark banana." has a chance of $\frac{1}{7}^2$ to occur. Another sentence; "draws banana a on banana." has a chance of $\frac{1}{7}^4 \cdot \frac{2}{7}$ to occur. In fact, $\frac{6}{7}^{th}$ of the generated sentences would not even start with "$Mark$". So. let us fix the first word such that the generated text does not start in the middle of a sentence. But it would have to be fixed for every sentence. A way to assure this, is to always let "$Mark$" follow "*banana.*". That means there should be a transition of probability 1 from the last word of the input text to the first word.

Improving this model further could be done by setting the chance of a word repeating itself to zero. In combination with that, if it were possible for it to measure wordclasses, it could assign larger probabilities to for example verbs after

names. But finding these classes is tricky. A model that looks at possibilities of combinations of word-classes might as well look at the combinations of words instead. This model should have similar benefits, as it should automatically prevent repeated words and lets the author of the input text worry about grammar.

So just randomly choosing words to generated a decent text is a terrible idea, but picking from sets of words that can follow certain other words may be a good idea.

3.2 A Better Model

The problem with the model in (6) is the total absence of grammatical structure. It is just a sequence of arbitrary words, which only depends on the set of words their assigned chances. So the model needs to be a bit 'aware' of grammar. This does not mean that grammar will be implemented into the model, but grammatical illness can be reduced to some extend by changing a core idea of model in (6). We can let the text decide what the odds for certain words are. Instead of determining the frequency of words, the model can compute the frequency of transitions from words to words. This is called a unigram model for words. As mentioned earlier, the first word will be fixed in this mdoel. For the same text used above, the transitions are:

" "	$\rightarrow$ "$Mark$"
"$Mark$"	$\rightarrow$ "$draws$"
"$draws$" and "on"	$\rightarrow$ "a"
"a"	$\rightarrow$ "$picture$" or "$banana.$"
"$picture$"	$\rightarrow$ "on"
"$banana.$"	$\rightarrow$ " "

Like mentioned before, the last word should have a transition to the first word. A proper random text generator should be able to generate more words than the input text contains. For this it needs that transition, or there might be a chance that the last state of the input text is an absorbing state. This also immediately makes the Markov Chain irreducible, since there is a chance that the input text will be generated twice in a row. So there is a finite path from any state in the first copy to any state in the second copy of the text. Creating the transition is forging data, but as the sample text gets larger, the impact it has on the text generating process will become smaller.

The previous transitions can be put in vector-matrix form. For a sentece that starts with "Mark", the model needs to choose "draws" next. So if for some t, $x_t = (100000)$, x_{t+1} has to be (010000). So P_{12} should equal 1 and all other P_{i2} should be 0. Doing this for all words yields

$$\begin{pmatrix} Mark \\ draws \\ a \\ picture \\ on \\ banana. \end{pmatrix}^T_{t+1} = \begin{pmatrix} Mark \\ draws \\ a \\ picture \\ on \\ banana. \end{pmatrix}^T_{t} \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

where the vector of words represent a state x_t, consisting of zeroes and a single 1. The words are shown to help visualise the process.

This way, the model always generates sentences that start with: "$Mark\ draws\ a$", leading to a fairly normal sentence or a loop of n times: "*picture on a picture on a picuture on a ...*" before finally ending with "*banana.*", which will happen with probability $\frac{1}{2}^{n+1}$.

So the grammatical structure is a lot better, but there are good chances that certain combinations of words are repeated. Also, the generated text is not very random, since there is only one random transition. The randomness will increase as the input text become larger. This will likely decrease the grammatical structure of the generated text.

In general it could be better to look at possible transitions of word combinations, meaning that the states would become pairs of words instead of single words. This is called a bigram model; the choice of the next state depends on the previous two states. The bigram model allows for more parts of sentence that were produced by the writer of the input text, so the sentences it produces should have improved grammatical structure.

In the test input text about Mark, all randomness would vanish for a bigram model, but for larger texts even the bigram model could become more random than desired. So the models will not only be able to generate text based on transitions between words and wordpairs, there will also the option to use wordtriples. The latter will be called the trigram model.

4 Making A Model

An important part of the model is finding the matrix P for arbitrary input texts. The first step is determining the **dictionary** of the input text.

Definition 4. *The dictionary for the model consists of all states that occur at least once in the subjected data.*

The dictionary is sorted alphabetically, case sensitive and punctuation sensitive. This means that the dictionary of words of the previous sentence would be: {The | alphabetically, | and | case | dictionary | is | punctuation | sensitive | sensitive.}

The dictionary is the same for all three models. But behind the scenes, the dictionary of wordpairs would be: {The dictionary | alphabetically, case | and punctuation | case sensitive | dictionary is | is sorted | punctuation sensitive. | sensitive and | sensitive. The | sorted alphabetically,}

There is a simple reason behind this stand-in dictionary. A bigram model for words is a unigram model for wordpairs and this way the scripts of the uni-, bi- and trigram can be nearly identical. Note that the dictionary for the bigram model contains a worpair that consists of the last and first word of the input text, because that transition was added. Titles and certain symbols such as quotation marks and asterisks can be ignored, as they do not influence the context of the input text. Sorting the dictionary alphabetically is not necessary, but it is done automatically by the command that is used and it does not harm the integrity of the program.

The second step is building P. The length D of the dictionary determines the size of P, it has to be a D×D matrix. Each row i represents the i^{th} state of the dictionary and each column j represents the j^{th} state of the dictionary. By definition, $P_{ij} = Pr(X_{t+1} = j \mid X_t = i)$, so all the entries P_{ij} contain the probabilities of transitions from states i to j. These probabilities will be the number of occurences of a transition (i, j), divided by the total number of transitions $\sum_k^D \#(i,k)$. Because most of the transitions do not occur, P will be a sparse matrix. The chosen probabilities for the transitions correspond to the maximum likelihood estimator.

Proof. By definition, an element P_{ij} of P is defined as $Pr(X_{n+1} = j \mid X_n = i)$. The likelihood of the input text is the chance of generating the input text, using P. Here, N is the number of words in the input text, D is the size of the dictionary, T denotes the input text, P denotes the transition matrix, Pr gives a probability and (a, b) is a transition between states a and b. The likelihood of T equals:

$$\begin{aligned} L_T(P) &= Pr(w_1) \prod_{t=1}^{N-1} Pr(w_t{+}1 \mid w_t) \\ &= .. \cdot P_{ij}^{\#(i,j)} \cdot .. \cdot P_{iD}^{\#(i,D)} \\ &= .. \cdot P_{ij}^{\#(i,j)} \cdot .. \cdot (1 - \sum_{k=1}^{D-1} P_{ik})^{\#(i,D)} \end{aligned}$$

The chance of generating a text is the chance of generating each word in the correct order. Note that we fixed the first word, so this chance becomes the multiplied probabilities of the transitions between those words. Multiplications commutate, so the product can be written with the powers of occurring transitions. Since a derivative is needed, the last word of the dicionary is expressed in terms of the other words. Now consider the loglikelihood:

$$\begin{aligned} l_T(P) &= log(L_T(P)) \\ &= \#(i,j) log(P_{ij}) + \#(i,D) log(1 - \sum_{k=1}^{D-1} P_{ik}) \end{aligned}$$

The derivative of the loglikelihood is taken (w.r.t. P_{ij}). For the maximum likelihood estimator, this derivative must be equal to zero.

$$\begin{aligned} \frac{\partial}{\partial P_{ij}} l_T(P) &= \frac{\#(i,j)}{P_{ij}} - \frac{\#(i,D)}{1 - \sum_{k=1}^{D-1} P_{ik}} \\ \frac{\#(i,j)}{P_{ij}} &= \frac{\#(i,D)}{1 - \sum_{k=1}^{D-1} P_{ik}} \\ P_{ij} &= \frac{\#(i,j)(1 - \sum_{k=1}^{D-1} P_{ik})}{\#(i,D)} \\ &= \frac{\#(i,j) \frac{\#(i,D)}{\sum_{k=1}^{D} \#(i,k)}}{\#(i,D)} \\ &= \frac{\#(i,j)}{\sum_{k=1}^{D} \#(i,k)} \end{aligned}$$

□

The third step is generating text with P. The first state was fixed, it is the first state of the input text. All other states are determined by their preceding state and P. Note that each row of P is a multinomial distribution. Suppose the model is generating text and it just completed generating the t^{th} state. State t corresponds to a state i of the dictionary, so row i contains the multinomial distribution which will be used to determine state $t+1$. This state corresponds to a state j of the dictionary. So state $t+2$ will be chosen from row j etc.

The final step is a compensation for the imitation of the bigram model of words by a unigram model of wordpairs. There is an unwanted side effect. Each pair of wordpairs: "word1 word2 - word2 word3" has a word in common, so the middle word occurs twice. This means that, apart from the first and last word, all words in the generated text are repeated. The solution to this problem becomes obvious when there are three worpairs: "word1 word2 - word2 word3 - word3 word4". The second wordpair can simply be skipped, because the first wordpair contains the first part and the third wordpair contains the second part of the second wordpair. This can be done for the entire text; every second wordpair at an even position will be skipped. For wordtriples, a similar solution exists. But in this case, the only wordtriples we do not skip are at a position 1 *mod* 3. Besides that, a small modification was made to make the generated text a bit more neat. The model checks for the last period, question- or exclamation mark of the generated text and cuts off the remainder so the result is a properly ended text.

4.1 Implementation In MATLAB

This section explains how the model was implemented in a programming language. We will be using MATLAB® for this, some commands that are available in this program will be mentioned. For different programming languages, it is likely that similar commands can be used. The preliminary steps are creating a function and defining the input arguments, like the input text.

The first step is creating a vector or something similar that contains the states of the input text and dictionary. The function starts by scanning the input text. This can be done with the **fscanf** command. The computer now knows what characters the input text contains. The next step is grouping those characters to form words. This can be done with a regular expression, using a **rexexp** command, in which a set of characters can be defined to be skipped or saved. With this regular expression, the input text becomes a cell array consisting of words. The next step is to get the dictionary from this cell array, which can be done with the **unique** command. This command automatically orders the words alphabetically, starting with capitulised words. After these steps, the function can build the transitionmatrix for the unigram model. Input arguments $n1$, $n2$ and $n3$ are added to allow the function to create transitionmatrices for the unigram, bigram or trigram models seperately, using some if commands. The steps above would normally be enough for all the models, but we imitate the bigram model of words with a unigram model of wordpairs, so additional steps must be taken. Note that we can use **circshift** command to permutate the input cell array. Now we may concatonate the input cell array with the permutated cell array, using the **strcat** command. This creates an alternative input cell array containing wordpairs, which will be used for the bigram model. Another circular shift and concatonation can be done to create the input cell array containing wordtriples. So all models have their own input cell array and a dictionary, but a single script can be used to build $P1$, $P2$ and $P3$.

The second step is building the transitionmatrices. The lengths a of the dictionary and b of the input cell array can be checked with a simple **length** command. The matrix has dimensions equal to the length of the dictionary. If $P = zeros(a, a)$ is used, the matrix has the correct size and most entries are already correct as well. A $fort = 1 : b - 1$ loop can be used to check the t^{th} state of the input cell array and this state can be compared with states from the dictionary, with a $fori = 1 : a$ loop. The check uses an if command and a **strcmp** command to compare strings. If the t^{th} state corresponds to the i^{th} state of the dictionary, there is a similar check for the $t+1^{th}$ state. If that state matches the j^{th} state of the dictionary, 1 is added to the corresponding entry: $P(i, j) = P(i, j) + 1$. After this has been

done for all states minus one, the transition from the last state to the first state is manually added with the same construction above. Now P is a countmatrix, it shows the number of occurences of all transitions. To make P rowstochasitc, the function normalises each row by multiplying P with a diagonal matrix. This diagonal matrix has the sums of the rows of P as entries. The sum can be taken with a **sum** command and a quick way of making a diagonal matrix, is using a sparse matrix, using a **spdiags** command.

The third step is generating text, this will be done with a seperate function. Because there are now two seperate functions, the important variables from the other function need to be defined as **global**. This allows the text generating function to call the matrices and dictionary lengths from the environment of the matrix function. The first things to define are the input arguments. It is important to know the desired length of the generated text and what model is to be used. A cell array with the desired length is made and the first state is fixed. This again uses the same construction to compare states from the dictionary to the input cell array, but a numerical value. is assigned. Then a numeric dictionary is made from the original dictioary, by giving the k^{th} state of the dictionary value k. Suppose that the first state of the input cell array is the i^{th} state of the dictionary, then the first value of the generated text is i and the function uses row i of P to determine the next state. That row is a multinomial distribution, so the **mnrnd** command can be used to pick a random column from it. This produces a vector consisting of zeroes and a single 1. Now the position of this 1 is checked with $forif$ construction. When it is found, the value of its position is assigned to the second cell of the cell array that will be the generated text. In the end, the cell array will be filled with numbers. All that remains to be done is convert the integers to the states they represent. But as an integer k corresponds to the k^{th} state of the dictionary, this is easy and will not be mentioned. For the bigram model, every second wordpair needs to be skipped and for the trigram model, both every second and third wordtriple need to be skipped to prevent duplications of words. Filling in the words is done with a for loop which takes steps of size 1 for the unigram model. For the bigram and trigram model it can simply take steps of 2 and 3 respectively. The generated text is written to a dummy file with the **dlmcell** command, which was downloaded seperately. That file is scanned and read as a regular expression. The characters in the file are read from the last character r to the first with a backward for loop: $fori = r : -1 : 1$. When the first period, question mark or exclamation mark is found, the remainder of the text is thrown away. These characters can simply be matched with a **strcmp** command in an if command. When a match is found the for loop is stopped by a **break** command. Then all of the characters to the right of that symbol are cut off with: $g(1, i+1 : r) = []$. Now the generated text is ended by a properly ended sentence.

The fourth step is improving the speed. Note that before this point, the functions do not use the fact that P is a sparse matrix, so some adjustments will be made. The MATLAB® works with sparse matrices is very different from normal matrices. Defining a sparse matrix is done by defining three vectors, which assign the location and value of nonzero elements. The first vector contains the row indices, the second vector holds the column indices and the third vector stores their values. The *sparse* command is used to create a matrix from these vectors. Example: $I = [1\ 3\ 1\ 2], J = [1\ 1\ 2\ 3], K = [3\ 6\ 8\ 7]$. The command $A = sparse(I, J, K, 4, 5)$ yields matrix A below, though it will never be shown like that.

$$A = \begin{pmatrix} 3 & 8 & 0 & 0 & 0 \\ 0 & 0 & 7 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

For the function that makes the transitionmatrices, this change is very small. The function defines I, J and K as vectors of the length of the input cell array instead of making a zero matrix P. Then it use the same code to fill in the entries of I, J and K. In that code we check for rows i, columns j and the k^{th} word of the input cell array. So the only line that needs to be replaced is $P(i, j) = P(i, j) + 1$. It is replaced by $I(k) = i, J(k) = j$ and $K(k) = K(k) + 1$. Then the function creates P with a sparse command like above, using the length of the dictionary as

dimensions.

The transition to a sparse format for the function that generates text requires some thought, because the *mnrnd* command does not work for the rows of a sparse matrix. So a new matrix p will be created from P, which contains the nonzero values of P. This can be done with a *nonzeros* command, but first the size of p needs to be determined. The amount of rows p has equals the amount of rows of P, because it still has to hold all the transitions. But for the amount of columns the function needs a vector z that has values corresponding to the amount of nonzero entries in the rows of P. From this vector it can take the maximum. Then p is created as a zero matrix and will be filled partially by letting the first entries be the nonzero entries of P. This process uses z to check the amount of nonzero entries per row. Now that p is determined, the *mnrnd* command can be used to choose a column j from a row of p. But this column corresponds to a state that does not correspond to the j^{th} state of the dictionary. To determine the following state, the function goes back to P and checks for the j^{th} nonzero entry. So it uses an if command in a for loop in a *while* loop so that the first $j-1$ nonzero entries of P are skipped. The if checks for nonzeroness, the for runs through the row of P and the *while* assures that the first $j-1$ states are skipped. This last step uses a **break** command to break the for loop that its index can be used as the assigned value in the generated text vector. The rest of the function needs no adjustments.

A last additional input argument was added, this is merely something that affects the **regexp** command so that it skips interpunction when the input text is a poem. The next section holds some examples of generated text.

4.2 Examples Of Generated Text

Consider a small input text:

Mark draws a picture on a banana. It is a picture of a baseball with batwings. He calls it a baseballbat.

The transitionmatrix for the unigram model is:

$$P1 = \begin{pmatrix}
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0\\
0&0&0&0&\frac{1}{5}&\frac{1}{5}&\frac{1}{5}&0&0&0&0&0&0&0&\frac{2}{5}&0\\
0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&\frac{1}{2}&\frac{1}{2}&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0
\end{pmatrix}$$

The transitionmatrix for the bigram model is:

$$P2 = \begin{pmatrix}
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&\frac{1}{2}&\frac{1}{2}&0\\
0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1\\
0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0\\
0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0
\end{pmatrix}$$

Consider some examples of generated text of at most length 40. The unigram model produces:

"Mark draws a picture of a picture on a baseballbat. Mark draws a picture of a banana. It is a banana. It is a baseballbat. Mark draws a banana."
"Mark draws a banana. It is a baseball with batwings. He calls it a picture of a banana. It is a baseballbat. Mark draws a picture on a picture of a picture of a picture on a picture of a..."

We see some entertaining sentences and their grammer appears to be correct. Note that the second example does not end with a period, it is simply a selection of generated text, but it points out the flaws of the unigram model. The bigram model produces:

"Mark draws a picture of a baseball with batwings. He calls it a baseballbat. Mark draws a picture on a banana. It is a picture on a banana. It is a picture of a baseball with batwings."
"Mark draws a picture on a banana. It is a picture on a banana. It is a picture on a banana. It is a picture of a baseball with batwings. He calls it a baseballbat."

The bigram model does not give the possibility to loop "picture of/on a picture..." like the unigram model does. But $P2$ shows only one state of randomness: it can choose "picture on" or "picture of" after "a picture" and all other transitions are fixed. So the question comes to mind: How much randomness is enough? The answer should depend on the size of the input text. For example, if the trigram model is used for the story about Mark and his banana, it would produce exact copies of the input text. But for bigger texts it will show plenty of randomness. This was merely a test text. Now the functions will be used for a book so that we may examine how well the models simulate the author.

5 Simulating An Author

Though this section seems to be mainly for entertainment purposes, it should provide some insights about what random text looks like. The generated text should be similar to the input text, since the model simulates the author. Decent looking parts of sentence will be in *italics*. Below are some generated sentences for "The Wonderful Wizard Of Oz" by L. Frank Baum. Here are two passages, using the unigram model:

"Doesn't anyone who was frightened, as a timid voice, and fast asleep. After climbing down and buttercups. Dorothy and sat down the magic was a beautiful country of the beast's head showed the fall away. As quick way past her do *but he lived in one spot, just under the truth.*"
"Lion said: It was so stepped upon him through the balloon and the Witch of the Lion and princes with three times, it also been carried *the travelers passed around the Lion's back to give me the girl to do* until you will be done, *he saw, standing silently gazing at all,* remarked the animals of her neck had been thinking again, and down my back to be able to fool would surely lost our promise, O Oz. Because you can do you like tigers."

The generated text appears to be gibberish at first sight. After a closer one can only conclude that indeed it is gibberish. The bigram model should produce better sentences:

"Dorothy lived in the North seemed to grieve the kind hearted Woodman, *or we may hurt these pretty little people so they believe I will tell you how grateful I am. Don't try, my dear,* you will help to you, *said the Scarecrow. I'll find a way. He then opened the big Lion she was riding in the rear of the forest* very thick on this side, *and it is my aunt who lives in the great beast in wonder, for he could raise his axe and* sat down."
"*This is strange, exclaimed Dorothy. The Lion went away into the air* and were so frightened that they can carry you over the swaying of the Kalidahs. I'm not sure about Kansas, said Oz, is made of tin nor straw, *and he was unable to bite Toto!*"

Some reasonable parts can be found in the sentences, but there is still much to be wished for. The text generated with the trigram model should be a lot more structured.

"*Dorothy lived in the clouds. The news spread rapidly throughout the city and everyone came to see the Great Oz to ask him for some brains. Oh, I see, said the Tin Woodman, as he felt his heart rattling around in his breast; and he told Dorothy he had discovered it to be a witch, had expected her to disappear in just that way, and was not surprised in the least. When Dorothy was left alone she began to feel hungry. So she went to the Throne Room and knocked at the door. Come in,* called Oz, and the Woodman both shook their heads, for they did not know what to

do with a heart if he had one. *I shall take the heart, returned the Tin Woodman; for brains do not make one happy, and happiness is the best thing* Dorothy can do is to travel to the Land of Oz, and two of them, *those who live in the City must wear spectacles night and day. Now they are all set free, and are grateful to you for having killed the Wicked Witch of the North* and South were good, *and I knew they would do for breakfast.*"

For a different book by a different author, the writing style should be different. So what was done above will be repeated for "The Call Of The Wild" by Jack London. The unigram model produces:

"And dreaming with the harness the bank ahead of the way broke the life abroad in any that he never came the evening and slept, or injured, had brought them as they prepared to attack, but all used, *the snow, where the beaten dogs fight* which is to crawl on end drop out of showing cruelly white moonlight."
"The hairy man understood Buck held on naked mountains *between him in Dave who had been devoured.* In quick flash of this bursting, rending, destroying, in terms, not know why, but so placatingly as that soars above his wrath and Hal into the red lolling tongue in which defied the previous December his head. No, it strange *and he realized that he said John Thornton.*"

Again, the unigram model produces gibberish. Perhaps the bigram model will show a difference in style:

This last *with a ferocious snarl he bounded straight up the slack and with nothing to do* was to command. But to prevent them from the wagon and started slowly on the trail by *the fierce invaders. Never had Buck seen such dogs. It seemed the ordained order of things that passes understanding. Buck heard them go and raised his head high, as though he was feeling too miserable to resist her,* taking it as though they were harnessing up, Dolly, who had never seen a sled to the bite of his previous departure."
He would lie in the morning. Likewise it was Thornton's privilege to knock *the runners which had been trembling abjectly, took heart at this open mutiny, and sprang upon Spitz.* But Francois, chuckling at the contact. *Every part, brain and body, nerve tissue and fibre, was keyed to the ground. He rapped his knuckles again as he came upon one of the wild,* come in from the standing forest."

These passages start to look like sentences, but they do not make too much sense either. The style does seem to be different from "The Wonderful Wizard Of Oz". The trigram model should again produce the best text:

"*Because men, groping in the Arctic darkness, had found a yellow metal, and because steamship and transportation companies were booming the find, thousands of men were rushing into the Northland. These men wanted dogs,* and the spark dimmed and *paled and seemed to lift with every movement, as though excess of vigor made each particular hair alive and active. Faithfulness and devotion, things born of fire and roof to the raw beginnings of life in the woods. For a day and a night he remained by the kill, eating and sleeping, turn and turn about. Then, rested, refreshed and strong, he turned his back and side.* He *had never been conspicuous for anything, went suddenly mad. She announced her condition by a long, heartbreaking wolf howl that sent every dog bristling with fear, then sprang straight for Buck. He had never seen an equal.*"

Again the trigram model produces the best looking sentences. The generated text consists of little passages from the book, with random jumps between them. This confirms that the functions work and it appears that this way of generating text has potential. But can it be used to distinguish between authors?

6 Authorship

Suppose there is a text of argueable origin and there are some candidate authors, a set $A = \{A_1 ... A_M\}$. Which author from this set is the most likely to have written the text? For this problem, we attempt to use a notion of distance called the Kulback-Leibler divergence. This gives the divergence from an author for a different author. The divergence of the candidate authors from the author of the unknown text will be calculated.

6.1 Kulback-Leibler Divergence

The Kulback-Leibler divergence is defined as

$$D_{KL}(P \mid Q) = \int log\left(\frac{dP}{dQ}\right) dP \tag{7}$$

for probability measures P and Q of which the transitionmatrices are subsets. The terms dP and dQ represent probability mass functions. The unknown author of the subjected text uses P and the candidate authors A_i use Q_i to write. Note that $D_{KL}(P \mid P)$ equals zero. So if Q is a very good estimate of P, the Kulback-Leibler divergence will be small. Though the Kulback-Leibler divergence was called a notion of distance, it is no metric. There is no symmetry in the divergence and the triangle inequality does not hold, but it is always positive (being only zero for $P = Q$). [3, p.55] The formula in equation (7) is an expectation:

$$E_P[log\left(\frac{dP}{dQ}\right)] = E_P[log(dP) - log(dQ)]$$

The linearity of the expectation can be used; the expectation of the sum is the sum of the expectations:

$$E_P[log(dP)] - E_P[log(dQ)]$$

So this is the expectation of P, based on dP and dQ. But P is unknown, so it can not be computed. In fact, it can not even be estimated, because it is also unknown know what kind of probability measure P is. But $E_P[log(dQ)]$ can be estimated. The only certainty we have, is that P writes his or her own texts. The set of all these texts will be called T. So:

$$E_P[log(dQ)] = \frac{1}{N}\sum_{i=1}^{N} log_{\,T_i}(dQ) \tag{8}$$

The subjected text t is only an element of this set, but it is the only known element of T. So (8) becomes:

$$log_{\,t}(dQ)$$

Because dQ is a probability mass function, this is the same as the likelihood of t, based on Q:

$$l_t(Q) = l_t(\hat{Q}) \tag{9}$$

Recall that a smaller Kullback-Leibler divergence means a better candidate and the divergence is always positive. So equation (9) needs to be maximized. Though Q_i is unknown, its estimate $\hat{Q}_i$ can be found with the random text generator that is supposed to simulate authors by using an input text that was written by A_i. This uses the maximum likelihood estimator, which maximizes the likelihood.

In the weird situation that there is an unknown author of which multiple texts are known, the following equation can be used.

$$\frac{1}{N}\sum_{T} l_t(Q)$$

There are some issues with this method. Possible issues are context and length. If the subjected text is an american history book and some of the candidates are estimated with books about the civil war, they will have an unfair advantage over the other candidates. The impact of context will be tested in a following section. Lenght is obvious, authors need to be represented by significant portions of text and this length should be similar for all candidates.

After calculating all these loglikelihoods, one candidate will be the most likely author. This one will be viewed as the actual author and likelihood ratio tests will be done with this author for the other candidates. This shows a third issue. What is the threshhold value? We can only speculate until the test results are known.

The last issue is a direct result of the method. Since ultimately, the test is based on transitions between n-tuples of words, the loglikelihoods immediately become zero for transitions in the text that are not for the candidates. For these missing transitions, a factor ε must be used instead of 0. The value of ε should likely depend on the size of the text.

Though, even if we solve these issues, the actual loglikelihood of the author writing his or her own text is still unknown, boundaries can be set. The lower boundary is the value of the best candidate author. The upper boundary will be the maximum likelihood estimator, for which 'selflikelihood' of t must be calculated. This selflikelihood is the loglikelihood of writing i when the transitionmatrix from t is used. The value that the selflikelyhood presents should be taken into consideration as it will have some impact on the verdict of the authorship. In essence, it will show how unlikely it is to write t even for the maximum likelihood estimator.

Implementing this as a function in MATLAB is quite easy, as the tricky parts can be found in the

other functions. So how it was done exaclty will not be mentioned. It only remains to solve the above-mentioned issues. Like previously mentioned, the third issue can not be solved yet. The first problems can be solved by choosing appropriate texts for the estimators of candidate authors. If no candidate has contextual overlap, there is no advantage over other candidates possible. The last issue can be solved by easily. In a function that was written for this authorship test, the number of missing transitions can be found. The number of words N of t is known, so there are $N-1$ transitions. So a sum term can be used for a candidate author that keeps track of the number of possible transitions. The rest r of the $N-1$ transitions are misses. A factor $r\ log(\varepsilon)$ is added to the loglikelhood to compensate for the missing transitions. But this leads to another issue about what ε should be.

6.2 Test Results

This section gives an idea whether the method works. If it does, it also gives an idea as to what value the threshhold should be. To check that the method works it is a bad idea to use a text of an unknown author, because that way it is impossible to compare the results of the test to what is true. So the test will be for "The Wonderful Wizard Of Oz" by L Frank Baum. The candidate authors and the books used to simulate their writing style are in the following table. Note that if one author is represented by multiple books, the author will be treated like several different authors.

Author	Abbreviation
L Frank Baum	LFB TWWOO
L Frank Baum	LFB TMOO
L Frank Baum	LFB TSF
Jack London	JLO TCOTW
J Branch Cabell	JBC DACOWW

N	The Wonderful Wizard Of Oz
O	The Magic Of Oz
V	The Sea Fairies
E	The Call Of The Wild
L	Domnei A Comedy Of Woman Worship

The following computations have been done with $\varepsilon = 0.01$, which is a very large factor. A smaller ε would be a lot more appropriate, but the verdict of the authorship of the book would remain unchanged, as the tolerance level should scale with ε. The loglikelihoods and misses for the entire book are shown below.

Author	unigram	bigram	trigram
Selflikelihood	-62478	-26717	-5503
LFB TMOO	-114939	-165715	-177777
JLO TCOTW	-138383	-174871	-179855
Misses	unigram	bigram	trigram
LFB TMOO	22119	35564	38588
JLO TCOTW	28348	37765	39050

The computation times for the previous results were huge, so future results will be for a part of the book. This part should still be a significant portion. All following results are for about a quarter of the entire book.

Author	unigram	bigram	trigram
Selflikelihood	-15855	-4326	-648
LFB TWWOO	-21179	-7806	-1530
LFB TMOO	-33270	-43332	-45855
LFB TSF	-34640	-43801	-45945
JLO TCOTW	-37826	-45234	-46302
JBC DACOWW	-36356	-45100	-46325
Misses	unigram	bigram	trigram
LFB TMOO	5817	9219	9947
LFB TSF	6170	9346	9970
JLO TCOTW	7353	9742	10053
JBC DACOWW	6985	9708	10059

Clearly the result for LFB TWWOO is unfair, because it can not miss transitions and it is not context free. The result for LFB TMOO is less unfair, but it is also about the land of Oz, so it is not context free. The result for LFB TSF is better than the other authors and it is fair. The most likely author of "The Wonderful Wizard Of Oz" is L Frank Baum, which is the actual author! Now that the best candidate (after scratching the unfair candidates) is known, the likelihood ratio tests can be done.

A likelihood ratio test usually tests two hypotheses and helps chosing bewteen them. The null-hypothesis H_0 is that the best candidate wrote the text. Our alternative hypotheses H_1 are that we can not be sure. If the ratio is smaller than an unknown tolerance level $0 < c < 1$, we discard H_0 and

if it is bigger, we accept it. [1]

$$\begin{aligned} \Lambda(x) &= \frac{L(\theta_0 \mid T)}{L(\theta_1 \mid T)} \\ log(\Lambda(T)) &= log\left(\frac{L(\theta_0 \mid T)}{L(\theta_1 \mid T)}\right) \\ log(\Lambda(T)) &= log(L(\theta_0 \mid T)) - log(L(\theta_1 \mid T))) \\ log(\Lambda(T)) &= l_{\theta_0}(T) - l_{\theta_1}(T) \end{aligned}$$

The results for the likelihood ratio tests are shown below. Note that these are actually loglikelihood ratios. The conclusion remains to be drawn.

LFB TSF	unigram	bigram	trigram
JBC DACOWW	1716	1299	380
JLO TCOTW	3186	1433	357

These numbers represent the estimated Kullback-Leibler divergence. The null-hypothesis is kept if $-log(c)$ is bigger than the divergence. But determining c is no trivial matter.

6.3 The Tolerance Level

The tolerance level is very complicated. First, a more appropriate value for ε (0.00001) will be used for the calculations. This yields the results below. Values that will not change are those of the selflikelihood, LFB TWWOO and the numbers of misses. These will not be repeated.

Author	unigram	bigram	trigram
LFB TMOO	-73452	-107015	-114567
LFB TSF	-77261	-108511	-114815
JLO TCOTW	-88618	-112530	-115746
JBC DACOWW	-84601	-112161	-115810

This leads to the following divergences.

LFB TSF	unigram	bigram	trigram
JBC DACOWW	7340	3650	995
JLO TCOTW	11357	4019	931

Even though it is unknown what exactly the tolerance level should be, the divergence from the actual author shown in the table is large for both candidates, so it is safe to say that the method works. The current ε is a thousand times smaller than its predecessor. In the logarithm that leads to roughly speaking a factor 3, which can be seen in the loglikelihood ratios. This is something to keep in mind. For an input text of different size, ε should be scaled accordingly. Suppose there are results for a large text and the next test results are for a tenth of the same text. Based on matching transitions, the large text has likelihood L_l and loglikelihood l_l, the smaller part has likelihood L_s and loglikelihood l_s. But there is also the portion of the likelihoods based on missing transitions: M_l, m_l, M_s and m_s. Suppose these are scaled directly by $5 \cdot \varepsilon_l = \varepsilon_s$. Here, m without subscript is the number of missing transitions for the smaller text. It is expected that:

$$\begin{array}{rclcrcl} L_s^{10} & = & L_l, & M_s & = & 5^m \cdot M_l \\ 10 \cdot l_s & = & l_l, & m_s & = & m \cdot log(5) + m_l \end{array}$$

This shows that scaling ε with the size of the text has no trivial connection to the loglikelihoods. This makes it harder to scale the tolerance level with ε.

Since the likelihoods for the candidate authors are closest to eachother for the trigram model; If the tolerance level is small enough for the trigram model, it will automatically be small enough for the other models, unless seperate tolerance levels are used for the different models. Note that in the table, the results show that a logtolerance of 100 is small enough. This means that the corresponding tolerance level would be $e^{-100} = 3.7 \cdot 10^{-42}$. This would mean that if the text was written roughly speaking $2.7 \cdot 10^{43}$ times, the odds are that one copy was written by a candidate author with distance 100 from the most likely candidate. Though this seems ridiculous, with $\varepsilon = 0.00001$ it would only come down to about 9 more misses. For a text with a length of roughly ten thousand words, these 9 misses do not seem to be that unlikely.

This leads to a strange dichotomy. On the one hand, we can not allow more misses for a trigram model, because the values are closer to eachother. One the other hand, we can not allow more misses for a unigram model than for a trigram model, because the unigram model will have less missing transitions.

To show another issue with the tolerance level, another book by L Frank Baum, named "The Life and Adventures of Santa Claus", was used to estimate his writing style. This estimate yields the following loglikelihoods for "The Wonderful Wizard Of Oz":

Author	unigram	bigram	trigram
LFB TLAAOSC	-80537	-110060	-115428
Misses	unigram	bigram	trigram
LFB TLAAOSC	6560	9517	10025

Though the results are better than those of the other authors, they worse than those of LFB TSF. This might also be the case because it is a smaller book. Therefore it is a good idea to scale ε with the size of the input text that is used to estimate an authors style. Still it poses a problem. If a single estimate of an author is accepted or refused, so should all of the other estimates. So the logtolerance needs to be large enough to allow some divergence for an author from him- or herself, but it needs to be small enough to be able to distinguish between different authors.

Possible solutions are taking the average of the likelihoods of all the estimates for the same author or using all of an authors books as a single input text to determine his or her writing style. The latter may be the better option, because that way there is one estimate per author. But more research needs to be done before claims can be made about the tolerance level. This will remain an open problem.

7 Conclusion

In this paper we attempted to do an authorship test, based on the theory of the Kullback-Leibler divergence. This divergence can not be computed, but with some estimation it can be reproduced in the form of a likelihood ratio test. The necessary loglikelihoods were estimated using a random text generator that was built to simulate authors, using a Markov Chain based on transitions between n-tuples of words. The assigned probabilities are maximum likelihood estimators, which makes them ideal for the authorship test. The issue of determining the tolerance level of the ratio tests remains unresolved, but the test results show that the method has potential. All the required computations were done with function that were built in MATLAB®. These can be found in the appendix. The texts that were used for testing the method were available thanks to Project Gutenberg.

The amounts of data used in this thesis could have been higher for more accurate results and the matters of the tolerance level and ε-scaling need further investigation. But the test results showed large divergences from the author for the other candidates, this is a satisfying result. Because the divergences are large, the logtolerance will very likely be smaller. Thus we must conclude that this method of authorship testing can be used to determine the authorship of texts.

References

[1] Directed from Wikipedia: A.M. Mood and F.A. Graybill. *Introduction to the Theory of Statistics*. 1963.

[2] Directed from Wikipedia: B.S. Everitt. *The Cambridge Dictionary of Statistics*. 2002.

[3] Directed from Wikipedia: C. Bishop. *Pattern Recognition and Machine Learning*. 2006.

[4] Directed from Wikipedia: Carl Meyer. *Matrix analysis and applied linear algebra*. 2000.

[5] Steven J. Leon. *Linear Algebra with applications*. Pearson.

[6] Frederick Mosteller and David L. Wallace. Inference in an authorship problem. *Journal Of The American Statistical Association*, 58:275–390, 1963.

[7] Mark Pollicot and Michiko Yuri. *Dynamical Systems and Ergodic Theory*. 1988.

[8] Prof. Alistair Sinclair. CS294 Markov Chain Monte Carlo: Foundations & Applications. `http://www.cs.berkeley.edu/~sinclair/cs294/n2.pdf/`. [Online; accessed July 2013].

[9] Wikipedia. Spectral Radius Theorem. `http://en.wikipedia.org/wiki/Spectral_radius#Theorem_.28Gelfand.27s_formula.2C_1941.29/`.

[10] Lance R. Williams. Markov. `http://www.cs.unm.edu/~williams/cs530/markov4.pdf/`. [Online; accessed July 2013].

A MATLAB Functions

A.1 Matrix Generator

```
function[TransitionMatricesSparse] = ProbMatS(inptext,n1,n2,n3,t)
%ProbMatS allows you to find transitionmatrices from given input texts
%Dummy output
TransitionMatricesSparse = 1;
% inptext = input text
%      n1 = 1 if you want to calculate P1
%      n2 = 1 if you want to calculate P2
%      n3 = 1 if you want to calculate P3
%       t = 'b' if inptext is a book
%       t = 'p' if inptext is a poem
%USE FOR EASE:  P = ProbMat('TestText.m',1,1,1);
%Use filename of the inputtext, example: inptext = 'TestText.m';
%Allow MATLAB to read the file
fileID = fopen(inptext);
%Read between spaces with c
inpstr = fscanf(fileID, '%c');
%==========================================================================
%CREATING MATRICES P
%==========================================================================
%We define important variables as global to use them later on
global P1 P2 P3 V1 V2 V3 D1 D2 D3 a c e
% P1 = Transitionmatrix:       word - word
% P2 = Transitionmatrix:   wordpair - wordpair
% P3 = Transitionmatrix: wordtriple - wordtriple
% V1 = input text: words
% V2 = input text: wordpairs
% V3 = input text: wordtriples
% D1 = Dictionary 1
% D2 = Dictionary 2
% D3 = Dictionary 3
%  a = length(D1)
%  c = length(D2)
%  e = length(D3)
%--------------------------------------------------------------------------
%==Prepare string and dictionary
%Seperate the input string in words by reading it as a regular expression
%\w = a-zA-Z0-9_ [include] [^exclude] \char = char \s = all whitespace
%For books we ignore certain characters
if t == 'b'
   V1 = regexp(inpstr,'\w*[\'']*[^\_\-\"\*\s]*','match');
end
%For poems we ignore punctuation
if t == 'p'
   V1 = regexp(inpstr,'\w*[\'']*[^\.\;\:\?\,\_\-\"\*\s]*','match');
end
%Dictionary 1: Unique words (case and punctuation sensitive)
D1 = unique(V1);
%Size of the input string V1
b = length(V1)
%P1 IS THE MATRIX FOR TRANSITIONS BETWEEN WORDS
%Don't waste time
if n1 == 1
```

```
%==Create transitionmatrix P1
%Size of dictionary D1
a = length(D1);
%Start measuring time
tic
%Make vectors of correct size
I = zeros(1,b);
J = I;
K = I;
%For each row i of P
for i=1:a
    %For each transition in V
    for k=1:b-1
        %If the k'th word of V is the i'th state of D
        if strcmp(V1{1,k},D1{1,i}) == 1
           %For each column j of P
           for j=1:a
               %If the k+1'th word of V is the j'th state of D
               if strcmp(V1{1,k+1},D1{1,j}) == 1
                  %Transition k->k+1 corresponds to P(i,j)
                  I(k) = i;
                  J(k) = j;
                  K(k) = K(k)+1;
               end
           end
        end
    end
end
%The last word has no transition, so we need to make it manually
%For each state of D
for i=1:a
    %If the last word of V is the i'th state of D
    if strcmp(V1{1,b},D1{1,i}) == 1
       %For each state of D
       for j=1:a
           %If the 1st word of V is the j'th state of D
           if strcmp(V1{1,1},D1{1,j}) == 1
               %Transition b->1 corresponds to P(i,j)
               I(b) = i;
               J(b) = j;
               K(b) = K(b)+1;
               break
           end
       end
    end
end
P1 = sparse(I,J,K,a,a);
%Make P1 stochastic by normalizing the rows
P1 = spdiags(1./sum(P1,2),0,a,a)*P1;
%Stop measuring time
disp('P1:')
toc
end
%-------------------------------------------------------------------------
%P2 IS THE MATRIX FOR TRANSITIONS BETWEEN WORDPAIRS
%Don't waste time
if n2 == 1
%==Prepare string and dictionary
%Generate new vector by cyclic permutation, ignoring the first word
Y1 = circshift(V1, [0,-1]);
```

```
%Generate new vector with spaces
Z = cell(1,b);
%For each entry of Z
for i=1:b;
    %The entry is a space
    Z{1,i} = ' ';
end
%Combine vectors V1, X and Y to create a cell array with wordpairs
V2 = strcat(V1,Z,Y1);
%Dictionary: Unique wordpairs (case and punctuation sensitive)
D2 = unique(V2);
%length(V1) = length(V2), length(D1) =/= length(D2)
c = length(D2);
%==Create transitionmatrix P2
%Start measuring time
tic
%Make vectors of correct size
I = zeros(1,b);
J = I;
K = I;
%For each row i of P
for i=1:c
    %For each transition in V
    for k=1:b-1
        %If the k'th word of V is the i'th state of D
        if strcmp(V2{1,k},D2{1,i}) == 1
           %For each column j of P
           for j=1:c
               %If the k+1'th word of V is the j'th state of D
               if strcmp(V2{1,k+1},D2{1,j}) == 1
                  %Transition k->k+1 corresponds to P(i,j)
                  I(k) = i;
                  J(k) = j;
                  K(k) = K(k)+1;
               end
           end
        end
    end
end
%The last word has no transition, so we need to make it manually
%For each state of D
for i=1:c
    %If the last word of V is the i'th state of D
    if strcmp(V2{1,b},D2{1,i}) == 1
       %For each state of D
       for j=1:c
           %If the 1st word of V is the j'th state of D
           if strcmp(V2{1,1},D2{1,j}) == 1
               %Transition b->1 corresponds to P(i,j)
               I(b) = i;
               J(b) = j;
               K(b) = K(b)+1;
               break
           end
       end
    end
end
P2 = sparse(I,J,K,c,c);
%Make P stochastic by normalizing the rows
P2 = spdiags(1./sum(P2,2),0,c,c)*P2;
%Stop measuring time
```

```
disp('P2:')
toc
end
%-------------------------------------------------------------------------
%P3 IS THE MATRIX FOR TRANSITIONS BETWEEN WORDTRIPLES
%Don't waste time
if n3 == 1
%==Prepare string and dictionary
%Generate new vector by cyclic permutation, ignoring the first word
Y1 = circshift(V1, [0,-1]);
%Generate new vector with spaces
Z = cell(1,b);
%For each entry of Z
for i=1:b;
    %The entry is a space
    Z{1,i} = ' ';
end
%Combine vectors V1, X and Y to create a cell array with wordpairs
V2 = strcat(V1,Z,Y1);
%Generate new vector by cyclic permutation, ignoring the first word
Y2 = circshift(Y1, [0,-1]);
%Combine vectors V2, Z and Y2 to create a cell array with wordtriples
V3 = strcat(V2,Z,Y2);
%Dictionary: Unique wordtriples (case and punctuation sensitive)
D3 = unique(V3);
%length(V1) = length(V3), length(D1) =/= length(D3)
e = length(D3);
%==Create transitionmatrix P3
%Start measuring time
tic
%Make vectors of correct size
I = zeros(1,b);
J = I;
K = I;
%For each row i of P
for i=1:e
    %For each transition in V
    for k=1:b-1
        %If the k'th word of V is the i'th state of D
        if strcmp(V3{1,k},D3{1,i}) == 1
           %For each column j of P
           for j=1:e
               %If the k+1'th word of V is the j'th state of D
               if strcmp(V3{1,k+1},D3{1,j}) == 1
                  %Transition k->k+1 corresponds to P(i,j)
                  I(k) = i;
                  J(k) = j;
                  K(k) = K(k)+1;
               end
           end
        end
    end
end
%The last word has no transition, so we need to make it manually
%For each state of D
for i=1:e
    %If the last word of V is the i'th state of D
    if strcmp(V3{1,b},D3{1,i}) == 1
       %For each state of D
```

```
        for j=1:e
            %If the 1st word of V is the j'th state of D
            if strcmp(V3{1,1},D3{1,j}) == 1
                %Transition b->1 corresponds to P(i,j)
                I(b) = i;
                J(b) = j;
                K(b) = K(b)+1;
                break
            end
        end
    end
end
P3 = sparse(I,J,K,e,e);
%Make P stochastic by normalizing the rows
P3 = spdiags(1./sum(P3,2),0,e,e)*P3;
%Stop measuring time
disp('P3:')
toc
end
%-----------------------------------------------------------------------
end
```

A.2 Text Generator

```
function[GeneratedTextSparse] = GeneTexS(N,n1,n2,n3)
%GeneTexS allows you to generate text using data from the ProbMat function
%Dummy output
GeneratedTextSparse = 1;
%  N = number of generated words    (+1 if N is an uneven number)
% n1 = 1 if you want to generate G1 (Generated text: words)
% n2 = 1 if you want to generate G2 (Generated text: wordpairs)
% n3 = 1 if you want to generate G3 (Generated text: wordtriples)
%USE FOR EASE:  G = GeneTexS(20,1,1,1);
%This function requires administrator rights for the dlmcell function
%=======================================================================
%GENERATING TEXT
%=======================================================================
%We call the data from the ProbMatS function
global P1 P2 P3 V1 V2 V3 D1 D2 D3 a c e
% P1 = Transitionmatrix:     word - word
% P2 = Transitionmatrix: wordpair - wordpair
% V1 = input text: words
% V2 = input text: wordpairs
% D1 = Dictionary 1
% D2 = Dictionary 2
%  a = length(D1)
%  c = length(D2)
%-----------------------------------------------------------------------
%Don't waste time
if n1 == 1;
%==The first word equals the first word of V1
%G1 is a numerical vector which represents the generated text
%Create a vector of correct size
G1 = zeros(1,N);
%For each state of D1
for i=1:a
    %If the first word of V1 is the i'th state of D1
```

```
    if strcmp(V1{1,1},D1{1,i}) == 1
        %The first word of G1 is the i'th state of D1
        G1(1) = i;
    end
end
%==The states in the numeric dictionary will be assigned values 1-a
%Create vector of correct size
d1 = zeros(1,a);
%Assign numerical values according to position
d1(1) = 1;
for i=1:a-1
    %The i'th position has value i
    d1(i+1) = d1(i)+1;
end
%==Generating the text
%Create a vector or maximum size
z1 = zeros(1,a);
%For each row of P
for i=1:a
    %The number of nonzero elements is checked
    z1(i) = length(nonzeros(P1(i,:)));
end
%Take the maximum of nonzero entries in a single row
M = max(z1);
%Create a matrix of correct size
p1 = zeros(a,M);
%For each row of p
for i=1:a
    %The nonzero entries of P are stored in the first columns
    p1(i,1:z1(i)) = (nonzeros(P1(i,:)))';
end
%For each transition
for t=1:N-1
    %For each state of D
    for i=1:a
        %If the t'th word of G is the i'th state of D
        if G1(t) == d1(i)
           %Row i of P is a multinomial distribution but P is sparse
           %Make a 0 0 1 0 0 type vector R by chosing randomly
           R = mnrnd(1,p1(i,:));
           %For each entry of R
           for j=1:M
               %If the entry equals 1
               if R(j) == 1
                  %Reset k
                  k = 0;
                  %Reset l
                  l = 0;
                  %Check for the entire row
                  %While k is not equal to j yet
                  while k < j
                        %For all entries we have not checked yet
                        for l=l+1:a
                            %If the entry is nonzero
                            if P1(i,l) ~= 0
                               %k gets closer to j
                               k = k+1;
                               break
                            end
                        end
                  end
                  %The t+1'th word of G is state j of p
                  %The j'th state of p has value l in D
```

```
                G1(t+1) = l;
            end
        end
        break
    end
  end
end
%Convert G to a cell array T
T1 = num2cell(G1);
%Detemine generated text by reverting numbers to words
%For each generated integer
for i=1:N
    %For each state of D
    for j=1:a;
        %If the i'th integer of G is j
        if G1(i) == j
           %The generated word equals the j'th state of D
           T1{i} = D1{1,j};
        end
    end
end
%Convert the cell array to a txt file
dlmcell('DummyText1.txt',T1,' ');
%==Removing text after last dot
remafdotstr1 = 'DummyText1.txt';
%Allow MATLAB to read the file
fileID1 = fopen(remafdotstr1);
%Read between spaces with c
remafdot1 = fscanf(fileID1, '%c');
%Find the number of characters in the string
r1=length(remafdot1);
%From the last character to the first
for i=r1:-1:1
    %If the character is a dot
    if (remafdot1(i) == '.')
       %Stop at that index
       break;
elseif (remafdot1(i) == '?')
       %Stop at that index
       break;
elseif (remafdot1(i) == '!')
       %Stop at that index
       break;
    end
end
%Remove the last entries
remafdot1(i+1:r1)=[];
%Write to a txt file
dlmwrite('GeneratedText1.txt',remafdot1,'')
end
%-------------------------------------------------------------------------
%Don't waste time
if n2 == 1;
%==The first wordpair equals the first wordpair of V2
%G2 is a numerical vector which represents the generated text
%Create a vector of correct size
G2 = zeros(1,N);
%For each state of D2
for i=1:c
    %If the first wordpair of V2 is the i'th state of D2
```

```
    if strcmp(V2{1,1},D2{1,i}) == 1
       %The first wordpair of G2 is the i'th state of D2
       G2(1) = i;
    end
end
%==The wordpairs in the numeric dictionary will be assigned values 1-c
%Create vector of correct size
d2 = zeros(1,c);
%Assign numerical values according to position
d2(1) = 1;
for i=1:c-1
    %The i'th position has value i
    d2(i+1) = d2(i)+1;
end
%==Generating the text
%Create a vector or maximum size
z2 = zeros(1,c);
%For each row of P
for i=1:c
    %The number of nonzero elements is checked
    z2(i) = length(nonzeros(P2(i,:)));
end
%Take the maximum of nonzero entries in a single row
M = max(z2);
%Create a matrix of correct size
p2 = zeros(c,M);
%For each row of p
for i=1:c
    %The nonzero entries of P are stored in the first columns
    p2(i,1:z2(i)) = (nonzeros(P2(i,:)))';
end
%For each transition
for t=1:N-1
    %For each state of D
    for i=1:c
        %If the t'th word of G is the i'th state of D
        if G2(t) == d2(i)
           %Row i of P is a multinomial distribution but P is sparse
           %Make a 0 0 1 0 0 type vector R by chosing randomly
           R = mnrnd(1,p2(i,:));
           %For each entry of R
           for j=1:M
               %If the entry equals 1
               if R(j) == 1
                  %Reset k
                  k = 0;
                  %Reset l
                  l = 0;
                  %Check for the entire row
                  %While k is not equal to j yet
                  while k < j
                        %For all entries we have not checked yet
                        for l=l+1:c
                            %If the entry is nonzero
                            if P2(i,l) ~= 0
                               %k gets closer to j
                               k = k+1;
                               break
                            end
                        end
                  end
                  %The t+1'th word of G is state j of p
                  %The j'th state of p has value l in D
```

```
                G2(t+1) = l;
            end
        end
        break
    end
  end
end
%Convert G2 to a cell array s2
s2 = num2cell(G2);
%Detemine generated text by reverting numbers to wordpairs
%For each generated integer
for i=1:N
    %For each state of D2
    for j=1:c;
        %If the i'th integer of G2 is j
        if G2(i) == j
           %The generated wordpair equals the j'th state of D2
           s2{i} = D2{1,j};
        end
    end
end
%==Preventing double words
%Make a new cell array, size depends on N being even or uneven
%If N is even
if mod(N,2) == 0
   %Size is N/2
   t2 = zeros(1,N/2);
%If N is uneven
elseif mod (N,2) == 1
       %Size is (N+1)/2
       t2 = zeros(1,(N-1)/2);
end
T2 = num2cell(t2);
%For each wordpair
for i=1:2:N
    %We skip each second wordpair
    T2{1,(i+1)/2} = s2{i};
end
%Convert the cell array to a txt file
dlmcell('DummyText2.txt',T2,' ');
%==Removing text after last dot
remafdotstr2 = 'DummyText2.txt';
%Allow MATLAB to read the file
fileID2 = fopen(remafdotstr2);
%Read between spaces with c
remafdot2 = fscanf(fileID2, '%c');
%Find the number of characters in the string
r2=length(remafdot2);
%From the last character to the first
for i=r2:-1:1
    %If the character is a dot
    if (remafdot2(i) == '.')
       %Stop at that index
       break;
elseif (remafdot2(i) == '?')
       %Stop at that index
       break;
elseif (remafdot2(i) == '!')
       %Stop at that index
       break;
    end
end
```

```
%Remove the last entries
remafdot2(i+1:r2)=[];
%Write to a txt file
dlmwrite('GeneratedText2.txt',remafdot2,'')
end
%------------------------------------------------------------------------
%Don't waste time
if n3 == 1;

%==The first wordtriple equals the first wordtriple of V3
%G3 is a numerical vector which represents the generated text
%Create a vector of correct size
G3 = zeros(1,N);
%For each state of D3
for i=1:e
    %If the first wordpair of V3 is the i'th state of D3
    if strcmp(V3{1,1},D3{1,i}) == 1
       %The first wordpair of G3 is the i'th state of D3
       G3(1) = i;
    end
end
%==The wordtriples in the numeric dictionary will be assigned values 1-e
%Create vector of correct size
d3 = zeros(1,e);
%Assign numerical values according to position
d3(1) = 1;
for i=1:e-1
    %The i'th position has value i
    d3(i+1) = d3(i)+1;
end
%==Generating the text
%Create a vector or maximum size
z3 = zeros(1,e);
%For each row of P
for i=1:e
    %The number of nonzero elements is checked
    z3(i) = length(nonzeros(P3(i,:)));
end
%Take the maximum of nonzero entries in a single row
M = max(z3);
%Create a matrix of correct size
p3 = zeros(e,M);
%For each row of p
for i=1:e
    %The nonzero entries of P are stored in the first columns
    p3(i,1:z3(i)) = (nonzeros(P3(i,:)))';
end
%For each transition
for t=1:N-1
    %For each state of D
    for i=1:e
        %If the t'th word of G is the i'th state of D
        if G3(t) == d3(i)
           %Row i of P is a multinomial distribution but P is sparse
           %Make a 0 0 1 0 0 type vector R by chosing randomly
           R = mnrnd(1,p3(i,:));
           %For each entry of R
           for j=1:M
               %If the entry equals 1
               if R(j) == 1
```

```
                    %Reset k
                    k = 0;
                    %Reset l
                    l = 0;
                    %Check for the entire row
                    %While k is not equal to j yet
                    while k < j
                          %For all entries we have not checked yet
                          for l=l+1:e
                              %If the entry is nonzero
                              if P3(i,l) ~= 0
                                 %k gets closer to j
                                 k = k+1;
                                 break
                              end
                          end
                    end
                    %The t+1'th word of G is state j of p
                    %The j'th state of p has value l in D
                    G3(t+1) = l;
                end
            end
            break
        end
    end
end
%Convert G3 to a cell array s3
s3 = num2cell(G3);
%Detemine generated text by reverting numbers to wordtriples
%For each generated integer
for i=1:N
    %For each wordpair in D3
    for j=1:e;
        %If the i'th integer of G3 is j
        if G3(i) == j
           %The generated wordpair equals the j'th state of D3
           s3{i} = D3{1,j};
        end
    end
end
%==Preventing double words
%Make a new cell array, size depends on N being divisible by 3
%If N is even
if mod(N,3) == 1
   %Size is (N+2)/2
   t3 = zeros(1,(N+2)/3);
%If N is uneven
elseif mod (N,3) == 2
       %Size is (N+1)/2
       t3 = zeros(1,(N+1)/3);
elseif mod (N,3) == 0
       %Size is N/3
       t3 = zeros(1,N/3);
end
T3 = num2cell(t3);
%For each wordtriple
for i=1:3:N
    %We skip each second and third wordtriple
    T3{1,(i+2)/3} = s3{i};
end
%Convert the cell array to a txt file
dlmcell('DummyText3.txt',T3,' ');
%==Removing text after last dot
remafdotstr3 = 'DummyText3.txt';
```

```
%Allow MATLAB to read the file
fileID3 = fopen(remafdotstr3);
%Read between spaces with c
remafdot3 = fscanf(fileID3, '%c');
%Find the number of characters in the string
r3=length(remafdot3);
%From the last character to the first
for i=r3:-1:1
    %If the character is a dot
    if (remafdot3(i) == '.')
       %Stop at that index
       break;
elseif (remafdot3(i) == '?')
       %Stop at that index
       break;
elseif (remafdot3(i) == '!')
       %Stop at that index
       break;
    end
end
%Remove the last entries
remafdot3(i+1:r3)=[];
%Write to a txt file
dlmwrite('GeneratedText3.txt',remafdot3,'')
end
%-------------------------------------------------------------------------
end
```

A.3 Authorship Test

```
function[X] = AuthorTestS(inptext,n1,n2,n3,r,t)
%AuthorTestS allows you to test authorship of the input text
%using the ProbMat function for author estimates
%Dummy output
X = 1;
% inptext = input text
%      n1 = 1 if you want to calculate P1
%      n2 = 1 if you want to calculate P2
%      n3 = 1 if you want to calculate P3
%       r = 1 if it is the first time you test the input text
%       t = 'b' if inptext is a book
%       t = 'p' if inptext is a poem
%USE FOR EASE:  A = AuthorTestS('TestText.m',1,1,1);
%Use filename of the inputtext, example: inptext = 'TestText.m';
%Allow MATLAB to read the file
fileID = fopen(inptext);
%Read between spaces with c
inpstr = fscanf(fileID, '%c');
%=========================================================================
%CREATING MATRICES C
%=========================================================================
%We call the data from the ProbMatS function
global P1 P2 P3 D1 D2 D3 a c e C1 C2 C3 S11 S12 S13
% P1 = Transitionmatrix:       word - word
% P2 = Transitionmatrix:   wordpair - wordpair
% P3 = Transitionmatrix: wordtriple - wordtriple
```

```
% V1 = input text: words
% V2 = input text: wordpairs
% V3 = input text: wordtriples
% D1 = Dictionary 1
% D2 = Dictionary 2
% D3 = Dictionary 3
%  a = length(D1)
%  c = length(D2)
%  e = length(D3)
%-------------------------------------------------------------------------
%==Prepare string and dictionary
%Seperate the input string in words by reading it as a regular expression
%\w = a-zA-Z0-9_, []= include, [^]= exclude, \char = char, \s = whitespace
%For books we ignore certain characters
if t == 'b'
   T1 = regexp(inpstr,'\w*[\'']*[^\_\-\"\*\s]*','match');
end
%For poems we ignore punctuation
if t == 'p'
   T1 = regexp(inpstr,'\w*[\'']*[^\.\;\:\?\,\_\-\"\*\s]*','match');
end
%Dictionary 1: Unique words (case and punctuation sensitive)
A1 = unique(T1);
%Size of the input string V1
b = length(T1);
%C1 IS THE MATRIX THAT COUNTS TRANSITIONS BETWEEN WORDS
%Don't waste time
if n1 == 1
%==Create countmatrix C1
%Size of dictionary D1
d = length(A1);
if r == 1
%Start measuring time
tic
%Make a matrix of correct size
I = zeros(1,b-1);
J = I;
K = I;
%For each row i of C
for i=1:d
    %For each transition in T
    for k=1:b-1
        %If the k'th word of T is the i'th state of A
        if strcmp(T1{1,k},A1{1,i}) == 1
           %For each column j of C
           for j=1:d
               %If the k+1'th word of T is the j'th state of A
               if strcmp(T1{1,k+1},A1{1,j}) == 1
                  %Transition k->k+1 corresponds to C(i,j)
                  I(k) = i;
                  J(k) = j;
                  K(k) = K(k)+1;
               end
           end
        end
    end
end
C1 = sparse(I,J,K,d,d);
```

```
%Stop measuring time
toc
%==It is not unimportant to check out the self-likelyhood
%Start measuring time
tic
%Make Q1 stochastic by normalizing the rows
Q1 = spdiags(1./sum(C1,2),0,d,d)*C1;
%The loglikelyhood of the text starts as 0
Sl1 = 0;
%For each state in A
for i=1:d
    %For each state in A
    for j=1:d
        %If there are transitions from i to j
        if C1(i,j) ~= 0
           %The likelyhood of those transitions equals the
           %probability of the transition to the power of its
           %occurrences.
           Sl1 = Sl1 + log(Q1(i,j)*C1(i,j));
        end
    end
end
%Stop measuring time
toc
end
%==Calculating the likelyhood of the input text using Q
%We need to match words i,j with Q(i,j), find its value in Q
%and raise it to the power C(i,j)
%Create tolerance level (prevent multiplications with 0)
epsilon1 = 10^(-5);
%Start measuring time
tic
%The loglikelyhood of the text starts as 0
l1 = 0;
%Create a sum term
S1 = 0;
%For each state in A
for i=1:d
    %For each state in A
    for j=1:d
        %If there are transitions from i to j
        if C1(i,j) ~= 0
           %For each state in D
           for k=1:a
               %If the i'th state of A equals the k'th state of D
               if strcmp(A1{1,i},D1{1,k}) == 1
                  %For each state in D
                  for l=1:a
                      %If the j'th state of A equals the l'th state of D
                      if strcmp(A1{1,j},D1{1,l}) == 1
                         if P1(k,l) ~= 0
                            %The likelyhood of those transitions equals the
                            %probability of the transition to the power of
                            %its occurrences.
                            l1 = l1 + log(P1(k,l)*C1(i,j));
                            S1 = S1 + C1(i,j);
                         end
                      end
                  end
               end
           end
        end
```

```
    end
end
%Stop measuring time
toc
%Calculate the number of missing transitions
m1 = b-1-S1;
%Add epsilon transitions for lost transitions
l1 = l1 + log(epsilon1)*m1;
end
%------------------------------------------------------------------------
%C2 IS THE MATRIX THAT COUNTS TRANSITIONS BETWEEN WORDPAIRS
%Don't waste time
if n2 == 1
%==Prepare string and dictionary
%Generate new vector by cyclic permutation, ignoring the first word
Y1 = circshift(T1, [0,-1]);
%Remove the last entry
Y1{1,b} = [];
%Generate new vector with spaces
Z = cell(1,b);
%For each entry of Z
for i=1:b-1;
    %The entry is a space
    Z{1,i} = ' ';
end
%Combine vectors T1, Z and Y to create a cell array with wordpairs
T2 = strcat(T1,Z,Y1);
%Dictionary: Unique wordpairs (case and punctuation sensitive)
A2 = unique(T2);
%length(T1) = length(T2), length(A1) =/= length(A2)
f = length(A2);
%==Create countmatrix C2
if r == 1
%Start measuring time
tic
%Make a matrix of correct size
I = zeros(1,b-1);
J = I;
K = I;
%For each row i of C
for i=1:f
    %For each transition in T
    for k=1:b-1
        %If the k'th word of T is the i'th state of A
        if strcmp(T2{1,k},A2{1,i}) == 1
           %For each column j of C
           for j=1:f
               %If the k+1'th word of T is the j'th state of A
               if strcmp(T2{1,k+1},A2{1,j}) == 1
                  %Transition k->k+1 corresponds to C(i,j)
                  I(k) = i;
                  J(k) = j;
                  K(k) = K(k)+1;
               end
           end
        end
    end
end
```

```
C2 = sparse(I,J,K,f,f);
%Stop measuring time
toc
%==It is not unimportant to check out the self-likelyhood
%Start measuring time
tic
%Make Q2 stochastic by normalizing the rows
Q2 = spdiags(1./sum(C2,2),0,f,f)*C2;
%The loglikelyhood of the text starts as 0
Sl2 = 0;
%For each state in A
for i=1:f
    %For each state in A
    for j=1:f
        %If there are transitions from i to j
        if C2(i,j) ~= 0
           %The likelyhood of those transitions equals the
           %probability of the transition to the power of its
           %occurrences.
           Sl2 = Sl2 + log(Q2(i,j)*C2(i,j));
        end
    end
end
%Stop measuring time
toc
end
%==Calculating the likelyhood of the input text using Q
%We need to match words i,j with Q(i,j), find its value in Q
%and raise it to the power C(i,j)
%Create tolerance level (prevent multiplications with 0)
epsilon2 = 10^(-5);
%Start measuring time
tic
%The loglikelyhood of the text starts as 0
l2 = 0;
%Create a sum term
S2 = 0;
%For each state in A
for i=1:f
    %For each state in A
    for j=1:f
        %If there are transitions from i to j
        if C2(i,j) ~= 0
           %For each state in D2
           for k=1:c
               %If the i'th state of A equals the k'th state of D
               if strcmp(A2{1,i},D2{1,k}) == 1
                  %For each state in D1
                  for l=1:c
                      %If the j'th state of A equals the l'th state of D
                      if strcmp(A2{1,j},D2{1,l}) == 1
                         if P2(k,l) ~= 0
                            %The likelyhood of those transitions equals the
                            %probability of the transition to the power of
                            %its occurrences.
                            l2 = l2 + log(P2(k,l)*C2(i,j));
                            S2 = S2 + C2(i,j);
                         end
                      end
                  end
```

```
            end
        end
    end
end
%Stop measuring time
toc
%Calculate the number of missing transitions
m2 = b-1-S2-1; %-1: last transition in T2 impossible in P2
%Add epsilon transitions for lost transitions
l2 = l2 + log(epsilon2)*m2;
end
%------------------------------------------------------------------------
%P3 IS THE MATRIX THAT COUNTS TRANSITIONS BETWEEN WORDTRIPLES
%Don't waste time
if n3 == 1
%==Prepare string and dictionary
%Generate new vector by cyclic permutation, ignoring the first word
Y1 = circshift(T1, [0,-1]);
%The last entry is empty
Y1{1,b} = [];
%Generate new vector with spaces
Z = cell(1,b);
%For each entry of Z
for i=1:b-1;
    %The entry is a space
    Z{1,i} = ' ';
end
%Combine vectors T1, Z and Y to create a cell array with wordpairs
T2 = strcat(T1,Z,Y1);
%Generate new vector by cyclic permutation, ignoring the first word
Y2 = circshift(Y1, [0,-1]);
%The last entry is empty
Y2{1,b} = [];
%Remove a space
Z{1,b-1} = [];
%Combine vectors T2, Z and Y2 to create a cell array with wordtriples
T3 = strcat(T2,Z,Y2);
%Dictionary: Unique wordtriples (case and punctuation sensitive)
A3 = unique(T3);
%length(T1) = length(T3), length(A1) =/= length(A3)
h = length(A3);
%==Create countmatrix C3
if r == 1
%Start measuring time
tic
%Make a matrix of correct size
I = zeros(1,b-1);
J = I;
K = I;
%For each row i of C
for i=1:h
    %For each transition in T
    for k=1:b-1
        %If the k'th word of T is the i'th state of A
        if strcmp(T3{1,k},A3{1,i}) == 1
```

```
            %For each column j of C
            for j=1:h
                %If the k+1'th word of T is the j'th state of A
                if strcmp(T3{1,k+1},A3{1,j}) == 1
                   %Transition k->k+1 corresponds to C(i,j)
                   I(k) = i;
                   J(k) = j;
                   K(k) = K(k)+1;
                end
            end
        end
    end
end
C3 = sparse(I,J,K,h,h);
%Stop measuring time
toc
%==It is not unimportant to check out the self-likelyhood
%Start measuring time
tic
%Make Q3 stochastic by normalizing the rows
Q3 = spdiags(1./sum(C3,2),0,h,h)*C3;
%The loglikelyhood of the text starts as 0
Sl3 = 0;
%For each state in A
for i=1:h
    %For each state in A
    for j=1:h
        %If there are transitions from i to j
        if C3(i,j) ~= 0
           %The likelyhood of those transitions equals the
           %probability of the transition to the power of its
           %occurrences.
           Sl3 = Sl3 + log(Q3(i,j)*C3(i,j));
        end
    end
end
%Stop measuring time
toc
end
%==Calculating the likelyhood of the input text using Q
%We need to match words i,j with Q(i,j), find its value in Q
%and raise it to the power C(i,j)
%Create tolerance level (prevent multiplications with 0)
epsilon3 = 10^(-5);
%Start measuring time
tic
%The loglikelyhood of the text starts as 0
l3 = 1;
%Create a sum term
S3 = 0;
%For each state in A
for i=1:h
    %For each state in A
    for j=1:h
        %If there are transitions from i to j
        if C3(i,j) ~= 0
           %For each state in D
           for k=1:e
               %If the i'th state of A3 equals the k'th state of D
               if strcmp(A3{1,i},D3{1,k}) == 1
```

```
                    %For each state in D3
                    for l=1:e
                        %If the j'th state of A equals the l'th state of D
                        if strcmp(A3{1,j},D3{1,l}) == 1
                           if P3(k,l) ~= 0
                              %The likelyhood of a transition equals the
                              %probability of the transition to the power of
                              %its occurrences.
                              l3 = l3 + log(P3(k,l)*C3(i,j));
                              S3 = S3 + C3(i,j);
                           end
                        end
                    end
                end
            end
        end
    end
end
%Stop measuring time
toc
%Calculate the number of missing transitions
m3 = b-1-S3-2; %-2: Last transitions of T3 impossible in P3
%Add epsilon transitions for lost transitions
l3 = l3 + log(epsilon3)*m3;
end
%------------------------------------------------------------------------
if n1 == 1
   X = [Sl1, l1, m1];
   Y = 'Sl1: %8.0f\t l1: %8.0f\t m1: %8.0f\n';
   fprintf(Y,X);
end
if n2 == 1
   X = [Sl2, l2, m2];
   Y = 'Sl2: %8.0f\t l2: %8.0f\t m2: %8.0f\n';
   fprintf(Y,X);
end
if n3 == 1
   X = [Sl3, l3, m3];
   Y = 'Sl3: %8.0f\t l3: %8.0f\t m3: %8.0f\n';
   fprintf(Y,X);
end

end
```