MINING HIDDEN MARKOV MODELS IN SEQUENCES OF CHARACTERS USING RECURRENT NEURAL NETWORKS

by

SUSHMITA KHAN

(Under the Direction of Ionut E. Iacob)

ABSTRACT

Restoring damaged historical manuscripts and making them available to the large public has been of great interest for humanities researchers long before computers provided assistance for this task. Current technologies and models make this process easier, more accurate, and capable of discovering parts that were previously unknown. We use Recurrent Neural Networks for uncovering hidden Markov models in sequences of characters from historic manuscripts. Such manuscripts are typically written in some archaic language, which makes the underlying machine learning problem inherently difficult, as not much training data is available, in general. We use bidirectional, hierarchical models for sequences of one or more characters, trained on the existent manuscript data. We tested our model and present experimental results using an Old English manuscript.

INDEX WORDS: Recurrent Neural Networks, Markov models, Character prediction

2009 Mathematics Subject Classification: 15A15, 41A10

MINING HIDDEN MARKOV MODELS IN SEQUENCES OF CHARACTERS USING RECURRENT NEURAL NETWORKS

by

SUSHMITA KHAN

B.Sc., Brac University, Bangladesh, 2013

MS., Georgia Southern University, 2018

MS., Georgia Southern University, 2020

A Thesis Submitted to the Graduate Faculty of Georgia Southern University in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

STATESBORO, GEORGIA

MINING HIDDEN MARKOV MODELS IN SEQUENCES OF CHARACTERS USING RECURRENT NEURAL NETWORKS

by

SUSHMITA KHAN

Major Professor:	Ionut E. Iacob
Committee:	Goran Lesaja
	Felix Hamza-Lup

Electronic Version Approved:
July 2020

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

Tabular environment:

$\mathbb{R}^n$	Real numbers
$M_{K,L}$	Text restoration model
$\mathcal{A}$	Alphabet list
$\overline{\mathcal{A}}$	Extended list of alphabet
$\mathcal{T}$	Complete manuscript text
S_j	Set of subsequence
7	And representation in old English
5	Missing character placeholder

CHAPTER 1

INTRODUCTION

Manuscripts and documentation existed in one form and language or the other for hundreds of years. Over the years, documents of wars, cultures, and fictional stories have been preserved and passed onto the next generation.

With time, language evolved branching out into many different ones, English being one of them. The English language has also evolved, with the primary difference lying in the alphabet set and representation of words. Many historic events were recorded in old English and preserved. However, over time the pages tore or decayed, the ink faded, leaving incomplete and incomprehensible words and even sentences.

This research aims to recover such a damaged manuscript using Recurrent Neural Networks and postulates that you can mine HMM sequences using the RNN model. I trained an RNN model that will predict the next characters, after inputs. Then I use the probabilities from the prediction in the RNN to mine the Markov model sequence. Both RNN and HMM falls under the general umbrella of pattern recognition. Although RNN and HMM work best with sequence, their optimal performance comes with recognizing the pattern of the sequence and using that knowledge to predict the next in the sequence.

1.1 PATTERN RECOGNITION

Pattern mining helps us extract graphs and sequences to develop model [5], such as the RNN. Pattern mining used in data mining and machine learning problems [5, 34]. Machine learning models like the RNN uses the features extracted through pattern mining to get more accurate and interpretable results [5]. Pattern mining algorithms include the apriori algorithm, prefixSpan, FP-tree, SPADE, etc. that can extract patterns from large sets of dat [34].

Pattern Mining is commonly used in text mining tasks to extract sequential patterns,

frequent itemsets, n-grams, etc [34]. Interesting sub-sequences from sequences are discovered by using techniques of sequential pattern mining [11]. Moreover, these interesting sub-sequences can be measured using an array of criteria as it occurs.

1.2 Hidden Markov Models

Hidden Markovian models are historically popular for sequence analysis. They have enjoyed widespread applications ranging from genome sequencing to speech recognition, much like RNN's in the present day. However, the training and performance capabilities of these two sequence analysis methods are disparate. Although RNN's are notoriously difficult to train, and requires a significantly large data, they are highly accurate when it comes to sequence prediction. On the flip side, HMM's are trained easily but do not always perform reliably.

In theory, a Hidden Markov Model(HMM) is a process that iterates over a finite number of states, while simultaneously generating a probability value about the most likely next character in the sequence. The name is hidden because the state transition sequences are hidden from the observer. To parameterize HMM, the matrix of transition probabilities is used between each state, and the output probability distribution is observed for signal frame [30].

1.3 Recurrent Neural Networks

Recurrent Neural Networks are a type of artificial neural network that can handle variable-length sequence input [9]. Their high-dimensional hidden state with nonlinear dynamics enables them to remember and process past information [26, 12]. The results are 10 generated as a probability distribution over the next element of the sequence, including cases with an input sequence of variable length [9]. These features make them lucrative for natural language tasks. Examples include generating texts [26] and sequence [12].

In recent years RNN has seen increased popularity in character and sequence generation due to its reliable performance and high accuracy. To predict the next character provided a sequence, we develop a character-level language model. Such a model operates to learn a probability distribution over the sequence. Based on this learning, it generates a probability distribution for the next character, and the character associated with the highest value is the most probable outcome.

Recurrent neural networks generate probability distribution to get the next character in a generated string. This probability distribution is the next input in the network [26]. Because of this phenomenon, RNN's are considered directed non-Markov models and resembles the sequence memorizatio [31]. However, both recurrent neural networks and markovian models, specifically hidden Markov models are sequence analysis techniques that share a common property. They can both take the context of the input into account and represent the context of the previous sequence [2].

In this research, we postulate that there exist hidden Markov models within the probabilistic prediction of Recurrent neural network models for predicting the next character. We use old English text from the Beowulf text, which is written in historic and old English. We train a recurrent neural network model with a Dense, SimpleRNN, and Embedding layer from Keras TensorFlow library to predict the next character in the sequence, given a set of characters as input.

Historic texts have missing characters either at the end or beginning of the sentence because they were scanned from old and decaying books with torn pages. The goal of this RNN is to learn based on the text provided, and predict the missing characters. The character with the highest prediction probability is the character that will belong in the missing spaces. Then, we also find the probability values of the next character using HMM. We compare the output of the two models, and we find that there exists a pattern of HMM within the RNN outputs.

The rest of this paper is as follows: chapter 2 enumerates on Hidden Markov Models. Chapter 3 talks about Recurrent Neural networks. I discuss the models and their use in chapter 4. Chapter 5 elaborates the results and finding of this study, and finally, the last makes the concluding remarks and future studies.

CHAPTER 2

HIDDEN MARKOV MODELS

Hidden Markov Models (HMM) are probabilistic models used in linear sequencing to generate a sequence [10]. The key concept of HMM is that it is a finite model that describes probability distribution over an infinite sequence [10]. HMM constitutes an input, a hidden and an output state such that, when we visit one state a residue is emitted according to the state's emission probability distribution. Based on this, we choose the next state. This state path is known as the Markov chain, implying that the next state that we move on to depends on the current state we are in. However, in HMM as we only have the observed sequence the underlying state path is hidden, thus bringing about the hidden component. These hidden states are the residue labels we will focus on inferring. In an HMM both the model parameters θ and the overall sequence "scores" $P(S,\pi \mid \text{HMM}, \theta)$ are all probabilities [10].

In essence, HMM invokes an unobserved or hidden state $s = s_1, \ldots, s_t, \ldots, s_T$ variables from the probability distribution over sequence of observations, where $y = y_1, \ldots, y_t, \ldots, y_T$ Basically, HMM is the sequence of the hidden states that has Markov characteristics, that is for a given s_t, s_τ is independent of s_ρ such that $\tau < t < \rho$ and y_t is independent of other variables given s_t. Such models have two parameters: the transition matrix whose it^{th} element is $P(S_{t+1} = j|s_t = i)$ and the emission matrix whose iq^{th} element is $P(Y_t = q|s_t = i)$. Baum-Welch algorithm, a special case of EM, estimates the parameters of HMM [1].

2.1 ARCHITECTURE OF HMM

If a given X_n and Y_n is discrete-time stochastic process and $n \geqslant 1$, the pair (X_n, Y_n) is considered HMM given X_n is a markov process and is not hidden and $P(Y_n \in A|X_1 = x_1, \ldots, X_n = x_n$. Note that X_n states are the hidden states and the emission states (also

known as output probability) is $P(Y_n \in A | X_n = x_n)$. The following image shows the architecture of an HMM:

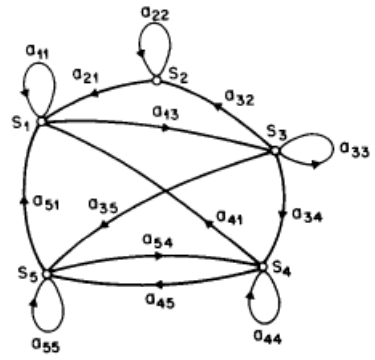


Figure 2.1: General architecture of instantiated HMM [23]

There is a hidden state x_t at given time t. The conditional probabilities for the hidden state depend on the value of the hidden variable x_{t-1}. However, it is worthy to note that the value at $t-2$ time does not influence the probability. This particular phenomenon is known as the Markov property. The observed variable y_t value depends on the hidden state x_t value. alue. HMM, 's have two parameters: the transition and the emission probabilities. The function of the transition probability is to control how the hidden state t is chosen given $t-1$.

We assume that the hidden state consists of N values and has N possible states where a hidden variable at a given time t can be in. There also exists transitional probability from this state to all other N states of the hidden variable at time $t+1$. We can determine only one transitional probability, given we know the others. Thus there is a total $N(N-1)$ transition parameters. Also, for each state, there is an emission/output probability that overlooks the distribution of the observed variable for a specific time given the state of the hidden variable. The emission probability's state's size depends on the type of the observed variable.

Over the years HMM's gained popularity with machine learning methods [10]. HMM's

14 are convenient because they can simply assumptions, thus making them successful [4]. HMM's are widely used in sequence data modeling tasks like speech recognition, bioinformatics protein, and DNA sequencing [10]. They are also used in text classification, segmentation, clustering, speech, and pattern recognition as well as parts of speech tagging [33].

Because of HMM's superior performance, they are widely used in processing sequence data like text data. [32] developed an HMM that can search for the best hypothesis provided in the text. Such a task needed the model to recognize the phonemes in a continuous stream. Drawing from concepts of speech recognition, in this scenario, they analyzed the transcript using HMM where the hidden states are the topic and the observations are words or sentences. An example of using HMM for text classification is demonstrated in[33]. The HMM model is built on the assumption of different sources that convey different aspects of information and yet can provide more general information on the topic. HMM has also been used to extract coded island information from natural language and source code. In such a process, the hidden states of HMM are used to model specific coded information [6].

CHAPTER 3

THE ARCHITECTURE OF A RECURRENT NEURAL NETWORK

Recurrent Neural Network(RNN) [25] is a type of artificial neural network which performs very well with sequence labeling task. RNN has an input layer x, a hidden layer h, and an output layer y. Input into an RNN is a sequence of vectors, and output is typically a sequence as well. In the simplest form of the RNN architecture, at a given time step *t*, the value of each layer is calculated using the following equations for hidden and output layers respectively:

$$h_t = f(Ux_t + Wh_{t-1}) \tag{3.1}$$

$$y_t = g(Vh_t) \tag{3.2}$$

where U, W, and V are the connection weight matrices in RNN, and $f(z)$ and $g(z)$ are sigmoid and softmax activation functions [22]. The rest of this chapter enumerates the calculations, sigmoid, and softmax functions in detail.

Although the basic architecture of RNN is like artificial neural networks, the difference is that it can send back feedback signals through the hidden layers to form a directed cycle [7]. RNN can use its internal memory to process the input of the timing sequence. Because the nodes directly connect with each other into a loop, it can access the internal state and show the dynamic timing behavior of the model. This property gives RNN an edge when handling input timing sequence thus making it an optimal choice for handle handwriting recognition and speech recognition.

RNN models a dynamical discrete-time system with input x_t, output y_t and hidden state h_t, where the subscript *t* represents time [21]. In essence, how an RNN function is that it receives an input at each time step, updates the hidden state, and makes a prediction [26]. A vanilla RNN has an input, hidden and a output layer. The hidden and output layers are mathematically represented below:

$$h_t = f_h(x_t, h_{t-1}) \tag{3.3}$$

$$y_t = f_o(h_t) \tag{3.4}$$

where f_h and f_o are state transitions. We construct RNN by defining the transition and output functions as follows:

$$h_t = f_h(x_t, h_{t-1}) = \phi_h(W^T h_{t-1} + U^T x_t) \tag{3.5}$$

$$y_t = f_o(h_t, x - t) = \phi_o(V^T h_t) \tag{3.6}$$

where x represents the input, h_t the hidden layers and y_t the output layers. U, W and V are the input, transition and output matrix and ϕ_h and ϕ_o are element wise non-linear functions [21].

Common choices for non-linear functions are logistic sigmoid function or hyperbolic tangent function ϕ_h as used in equation 3.1. We add the non-linear hyperbolic tangent function coordinate wise followed by a bias value in each layer. The following equation enumerates the application of the hyperbolic tangent function and a bias value:

$$h_t = tanh(W_{hx} + W_{hh} h_{t-1} + b_h) \tag{3.7}$$

where W_{hx} is the input to hidden value, W_{hh} is the hidden-to-hidden value, t is the time occurrence and b_h is the bias value [26].

Between the hidden and the output layer, we use a softmax layer to generate the prediction probabilities, which is the output from an RNN [14]. The following equation represents the value of one node between the hidden and the output layer as shown in [26]:

$$Y_t = W_{yh} h_t + b_o \tag{3.8}$$

where W_{yh} is the hidden to output weight matrix, and b_o is the bias. Then we apply the softmax function into the sum of the matrices as shown below [14]:

$$y_t = softmax(\sum_{t=1}^{T} (W_t) h_t) \tag{3.9}$$

The following is the formal definition of a $softmax()$ followed by a type of $softmax()$ called $Relu()$.

Definition 1. [Softmax function] The $softmax$ function $softmax : \mathbb{R}^n \to (0,1)^n$ is defined as:

$$softmax(x_1, ..., x_n) = (s_1, ..., s_n)$$

where

$$s_i = \frac{e^{x_i}}{\sum_{i=1}^{n} e^{x_i}}$$

Definition 2. [Relu function] The $Relu$ function $Relu : \mathbb{R} \to [0, \infty)$ is defined as:

$$Relu(x) = \begin{cases} 0, & \text{if } x \leq 0 \\ x, & \text{if } x > 0 \end{cases}$$

The $relu$ function can be naturally extended in the Euclidian space as follows:

$$Relu : \mathbb{R}^n \to [0, \infty)^n, \quad Relu(x_1, ..., x_n) = (Relu(x_1), ..., Relu(x_n))$$

The purpose of the $softmax()$ function is to normalize any vector in $\mathbb{R}^n$ to a vector of probabilities of which sum is 1. The $Relu()$ function is the current, more practical replacement of nonlinear transformation functions $sigmoid$ and $hyperbolic\ tangent$. $Relu()$ is chacterized by its simplicity (easy to compute), linear behavior, and output sparsity and produces excellent practical results.

Most RNN learns by backpropagation and backpropagatation through time. Through back-propagation, the gradient of RNN can be calculated [25, 29]. Backpropagation occurs after the forward propagation is completed and there is an predicted output. The network calculates the difference between the actual and predicted output known as the loss function. Then the network moves backward through the hidden layers. As it goes backwards, it calculates the gradient of the loss function and then adjusts the weights associated with each node. This adjustment contributes to improvement predicted output. As backpropagation involves calculating the gradient, it is also known as a type of gradient descent.

RNN use another mechanism to finish training. It is called backpropagation through time. Note that RNN's share weights across all layer, and the nodes are connected with

itself. So, during backpropagation, the gradient for each node is calculated across all the time steps, and the weights are adjusted. This phenomenon is known as backpropagation through time. The key difference between backpropagation and backpropagation through time is that the latter sums the gradient of each node for all time steps. Note that this happens for each node in each layer. However, this brings the issue of vanishing or exploding gradient that various studies propose different ways of handling. A popular choice is using LSTM [15, 13]. Often times studies used stochastic gradient descent to deal with the issue of extremely large gradients [14].

Finally, RNN's has one more component: the cost function. In essence, the cost function is the difference between the actual output and the predicted output. For performance accuracy, it is essential to minimize this loss function. So, for a given sequence,

$$D = (x_1^n, y_1^n, ..., x_{T_n}^n, y_{T_n}^n)_{n=1}^N \tag{3.10}$$

The output parameters are estimated by the RNN by minimizing the following cost function:

$$J(\theta) = 1/N \sum_{n=1}^{N} \sum_{t=1}^{T_n} d(y_t^n, f_o(h_t(n) \tag{3.11}$$

where $h_t^n = f_h(x_t^n, h_{t-1}^n)$ and $h_0^n = 0$ and $d(a, b)$ is a measure like cross-entropy.

RNN has been used for various sequence learning tasks because of its efficacy with sequence analysis and prediction. Because of these properties RNN [25] has recently become a popular choice for modeling variable-length sequences. They have been applied in many domains, but have been particularly popular with language tasks. As they have high dimensional hidden state and non-linear evolution, it gives them abundant expressive power. This also gives the hidden state opportunity to integrate information with multiple time steps. These collective processes are used to make the predictions accurate[26]. Thus RNN has garnered its popularity. Some notable examples of language modeling includes the works of [12, 21, 19, 26].

To accomplish complex language modeling tasks, simple RNN's are extended to deep RNN's (DRNN). This means that they have multiple recurrent hidden layers stacked on top of one another because much more effective at representing some functions [3]. An example of deep RNN is Long-Short Term Memory (LSTM). DRNN combats the RNN's weakness by introducing the concept of a multi-layer perceptron such that the hierarchical processing of inputs occurs through multiple layers. Deep RNN has temporal feedback loops in each layer. With each network update, the new information moves up the hierarchy and temporal context is added[14]. Deep RNN is integral in sequence and particular language modeling because through the iteration over the hidden layers, and the ability to grasp context over a time state, they can often accurately predict text sequences provided input.

A deep RNN or DRNN has L number of layers with each having N neurons. For example, with a time-series sequence of $s(t)$ as the input of N dimension and a target output sequence of $y^L(t)$. The DRNN backpropagates through time to update the weights of the hidden networks for the number of epochs provided by the user. In such a DRNN architecture, it is typical to see the fading memory of the input in the bottom layer. Furthermore, in the next layer, we can expect to see the fading memory of the hidden state. So, for each additional layer, the fading memory of the input reaches further in the past.

Because of RNN's performance capability with sequence labeling, it is popularly used in character-level language modeling. In such tasks, it is typical of the model to predict the next character, given a sequence. To put it formally, for a given training sequence $x_1.....x_T$, such a model will use it's output sequence of $o_1, ..., oT$ to calculate the probability of the predicted sequence $P(x_{t+1}, jx_t = softman(o_t))$, where softmax is defined by $P(softmax(o_t = j)) = exp(o(j))$.

The objective of a language model is to maximize the log probability of the training sequence to predict the most like next character in the sequence. In this study, we cap-

italize on such features of RNN. Instead of the maximum log probability, we output all the possible prediction probability from the RNN, with the ultimate goal of using these prediction probabilities to mine hidden Markov models. For this study, we have built two deep RNN with 128 and 256 hidden layers of the Embedding layer, SimpleRNN layer with relu activation, and dense layer with softmax activation using the Keras library. The first DRNN model trains on the old English Beowulf corpus, and for a given character predicts the probability of the next characters. The second DRNN model trains on the same corpus but for a given input sequence of length n, it predicts the probabilities of the $n-1$ characters following the n sequence input, where n is a variable provided by the user. Both the models output a table of the probabilities of the predictions in ascending order.

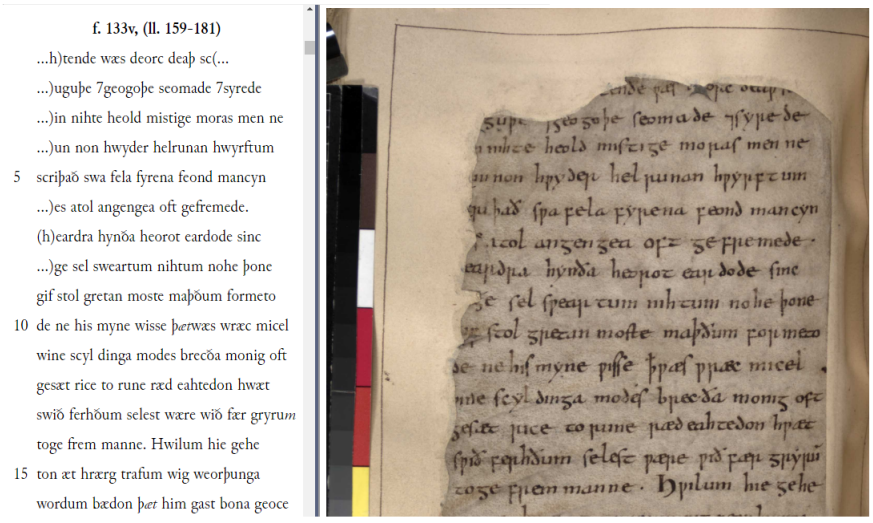
CHAPTER 4

FINDING HIDDEN MARKOV SEQUENCES USING RECURRENT NEURAL NETWORKS

In this chapter, we describe the structural architectures of a Hidden Markov Model (HMM) and a Recurrent Neural Network we use to predict sequences of characters. For this purpose, we use a short fragment of text from the Electronic Beowulf [18] manuscript, which is written in Old English. To our knowledge, no prior studies were performed on character frequencies and automatic prediction of character sequences was performed on Old English texts (as opposed to English, in general, and/or possibly other current languages). However, the applicability of our study and methods can be extended to sequences of symbols based on any alphabet.

Our choice of using text from an Old English manuscript has two motivations. Firstly, the study of such manuscripts is of great interest for humanities and widely popular in classrooms [16, 17]. Secondly, a typical search tool may reveal little information about such texts, as they use some special characters that make them "non-standard" both in terms of the appearance/encoding of such characters and the frequencies these characters occur in the text. Also, our fist motivation gives us a practical interest in tacking the problem of character sequence prediction in Old English manuscripts. Such historical documents are typically in poor shape (as illustrated, for instance, in Figure 4.1 right) and hardly accessible for the general public (they stored away from the large public, in special collections of libraries and museums). Practical tools for the analysis of such documents are scarce but of great interest.

The rest of this chapter is organized as follows. We start be formalizing the problem we study in Section 4.1. We introduce the specific architectures of the Recurrent Neural Network (RNN) and Hidden Markov Model (HMM) we are using in Sections 4.2 and 4.3, respectively.



f. 133v, (ll. 159-181)

...h)tende wæs deorc deaþ sc(...
...)uguþe 7geogoþe scomade 7syrede
...)in nihte heold mistige moras men ne
...)un non hwyder helrunan hwyrftum
scriþaþ swa fela fyrena feond mancyn
...)es atol angengea oft gefremede.
(h)eardra hynða heorot eardode sinc
...)ge sel sweartum nihtum nohe þone
gif stol gretan moste maþðum formeto
de ne his myne wisse þærwæs wræc micel
wine scyl dinga modes brecða monig oft
gesæt rice to rune ræd eahtedon hwæt
swið ferhðum selest wære wið fær gryrum
toge frem manne. Hwilum hie gehe
ton æt hrærg trafum wig weorþunga
wordum bædon þæt him gast bona geoce

Figure 4.1: A folio of the original Beowulf manuscript [18]

4.1 Preliminaries and the problem statement

The work of transcription and restoration of manuscripts text (like the one in Figure 4.1 right) is performed manually by highly trained scholars. The result of the transcription (Figure 4.1 left) is often affected by missing characters (represented by dots in parentheses in Figure 4.1 left), some of which are later recovered by experts after tedious analysis. Our work aims to assign some quantitative measures to restore the missing parts of the text. More specifically, we create a model capable of learning from the text that can be clearly recovered and subsequently making predictions for the missing parts. The model predicts the missing parts with certain probabilities, providing content restoration alternatives with probabilities for each option.

We start by introducing the list of character symbols in our study. We denote by $\mathcal{A}$ the set of all characters that appear in the manuscript we analyze (the alphabet):

$$\mathcal{A} = \{7, a, æ, b, c, d, e, ð, f, g, h, i, l, m, n, o, p, r, s, t, þ, u, w\} \tag{4.1}$$

We note that the symbol "7" is a specific Old English symbol denoting the conjunction

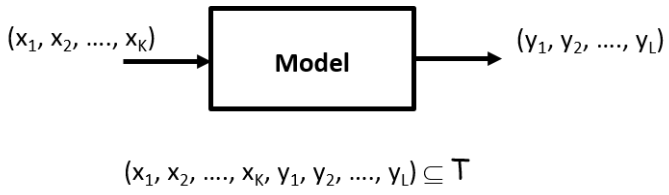


Figure 4.2: The text restoration model

"ond" (in Old English), that is, "and" (in modern English). For a character $c \in \mathcal{A}$ we denote by $indexc$ the index of the character c in the alphabet $\mathcal{A}$ (an integer from 1 to $|\mathcal{A}|$), and by $index([c_{i_1}, ..., c_{i_k}])$ the sequence of all indices of characters in the sequence $[c_{i_1}, ..., c_{i_k}]$. If i is the index of character $c_i \in \mathcal{A}$, we write $\mathcal{A}[i] = c_i$ (the alphabet's character at index i is c_i).

We extend this alphabet with two characters, the space character and the "missing characters placeholder, '5'. We denote the extended alphabet by $\overline{\mathcal{A}} = \textvisiblespace, 5 \cup A$.

We next introduce some notations that will help us formalize the problem we tackle in this study. The original, complete manuscript text we plan to analyze is denoted as $\mathcal{T} = [c_i]_{1 \leq n}$ and represents a sequence of symbols in the alphabet (4.1) plus the space symbol: $c_i \in \mathcal{A} \cup \{\textvisiblespace\}$. For convenience, we do not include any punctuation in our text[1]. We let $n = len(\mathcal{T})$ denote the length of sequence $\mathcal{T}$. Similarly, the *actual* (recovered) manuscript text is denoted as $\mathcal{R}$ and represents a sequence of symbols from the alphabet (4.1) plus the space symbol and a placeholder '5' for missing characters: $\mathcal{R} = [c_i]_{1 \leq m},\ c_i \in \overline{\mathcal{A}}$. We let $m = len(\mathcal{R})$ denote the length of the sequence $\mathcal{R}$. Note that the missing characters placeholder '5' is, in general, a replacement for more than one character missing in the recovered manuscript text $\mathcal{R}$. That is, in general, $len(\mathcal{T}) > len(\mathcal{R})$. Informally, our task is building a model capable of learning from the recovered manuscript sequence $\mathcal{R}$ and make predictions of actual character(s) in $\mathcal{A}$ each occurrence of '5' might stand for. We let $[x_1, x_2, ..., x_k] \subseteq \mathcal{T}$ denote a subsequence in $\mathcal{T}$ (and similarly for a subsequence in $\mathcal{R}$).

[1]The original manuscript only includes periods, but not commas or other punctuation marks

Formally, the problem of recovering the original manuscript text can be formulated as follows.

Definition 3. [The text restoration problem] The text restoration problem consists of finding the best replacements for the placeholders '5' in $\mathcal{T}$ using the information in the recovered text $\mathcal{R}$.

We note that the "best replacements" as stated in the definition is not equivalent to the "correct replacements" as the solution to the problem will always include some degree of uncertainty. In our work, we aim to solve the problem automatically, by building models capable of finding possible solutions to the problem. For this purpose, we define a model for solving the problem as follows.

Definition 4. [Text restoration model] Let S_J be the set of all subsequences of some length J of $\mathcal{R} - \{5\}$. A text restoration model $M_{K,L}$ is a mapping $S_K \xrightarrow{M_{K,L}} S_L$ such that

$$M_{K,L}\left([x_1, x_2, ..., x_K]\right) = [y_1, y_2, ..., y_L]$$

where $[x_1, x_2, ..., x_K, y_1, y_2, ..., y_L] \subseteq \mathcal{T}$.

In other words, a model is a mapping that takes a sequence $[x_1, x_2, ..., x_K]$ and predicts the following sequence $[y_1, y_2, ..., y_L]$ in the original manuscript text. Figure 4.2 illustrates the transformation performed by the model. Notice that a model explicitly excludes the placeholder '5' both from the input set of values and the output set of values. This makes the problem we present here different from the typical sequence-to-sequence prediction problem (as in [27], for instance). Moreover, forcing a model deal only with the alphabet characters makes building the model more difficult, as the recovered manuscript text $\mathcal{R}$ (which is typically used to create an accurate model) is fragmented into subsequences of various lengths after the placeholder character '5' is being removed.

A quick observation related to the model in Definition 4 is that the model is not unique: to any given input sequence one or more correct output sequences can follow. This raises

the question of confidence in a model (which is related to the “best replacement" statement in Definition 3). For this purpose, we define the confidence of the model output as follows.

Definition 5. [Confidence of model output] The confidence of a model $S_K \xrightarrow{M_{K,L}} S_L$ output is the conditional probability

$$P\left(y_1, y_2, ..., y_L | x_1, x_2, ..., x_K\right), \text{ where } [y_1, y_2, ..., y_L] = M_{K,L}\left([x_1, x_2, ..., x_K]\right)$$

In practice, a more useful way to build models for text recovery is by computing a few possible outcomes together with their confidence values. Consequently, we define next the *Multiple Outputs Text Restoration Model* (MOTRM) as a practical enhancement of the model in Definition 4.

Definition 6. [MOTRM] Let S_J be the set of all subsequences of some length J of $\mathcal{R}-\{5\}$. A multiple outputs text restoration model $M_{K,L}^{(n)}$ is a mapping $S_K \xrightarrow{M_{K,L}} (S_L \times (0,1))^n$ such that

$$M_{K,L}^{(n)}\left([x_1, x_2, ..., x_K]\right) = \Big\{([y_1^{(i)}, y_2^{(i)}, ..., y_L^{(i)}], P^{(i)}) \mid [x_1, x_2, ..., x_K, y_1^{(i)}, y_2^{(i)}, ..., y_L^{(i)}] \subseteq \mathcal{T},$$

$$P^{(i)} = P\left(y_1^{(i)}, y_2^{(i)}, ..., y_L^{(i)} | x_1, x_2, ..., x_K\right), i = 1, ..., n\Big\}$$

(That is, a set of possible sequences in $\mathcal{T}$ together with their confidence values.)

The next sections will describe two practical implementation of the MOTRM using Recurrent Neural Networks (RNNs) and Hidden Markup Models (HMMs).

4.2 The Recurrent Neural Network Model for character sequence prediction

Recurrent Neural Network (RNN) models rely on early work on Neural Networks [24] and have recently specialized, among a broad set of applications, in sequence to sequence learning applications [27]. In a nutshell, *sequence to sequence learning* consists of building

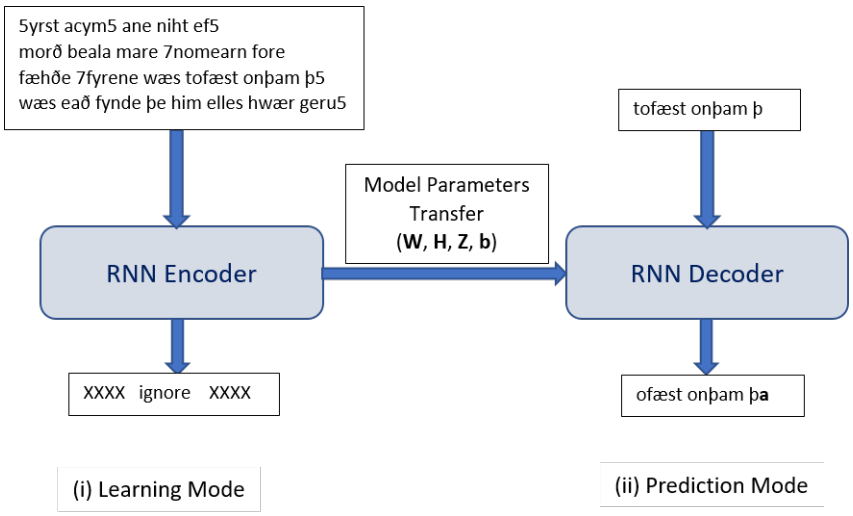


Figure 4.3: The RNN sequence-to-sequence learning model

a model capable of converting a sequence from a domain (input sequence domain) into another sequence from another domain (output sequence domain). The input and output sequences can have different lengths and the input and output domains can be the same or different. A classical example of a sequence to sequence learning problems is the problem of translating from one language (the input domain, such as the English vocabulary) into another language (the output domain, such as the French vocabulary).

The sequence to sequence learning problem typically comes in two versions: (i) equal size input and output and (ii) different sizes for input and output sequences. The former version of the problem is less general and typically applies encoding-decoding sequences of the same length in the same domain. Typically, the models in this category are easier to implement and more accurate. The later version is more general (but more complex) and it is being used in a variety of applications: translations, text-to-speech, and speech-to-text conversions, image recognition, etc. Our manuscript text restoration problem, as define in the previous section, falls in the second category of sequence to sequence learning problems: while the seek conversions between sequences of characters over the same alphabet, the input, and output sequences have different lengths.

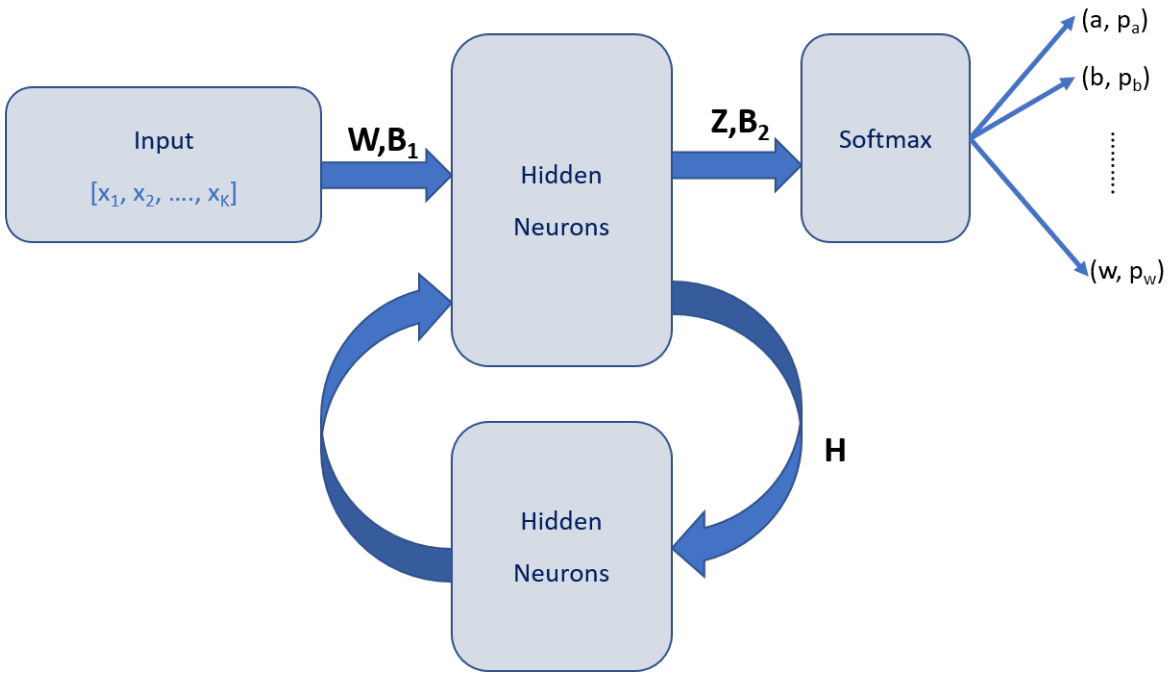


Figure 4.4: RNN model $M_{K,1}^{(26)}$

A typical RNN model for sequence-to-sequence learning and classification is illustrated in Figure 4.3. This model is built in two phases. In the first phase, the model parameters are optimized using a known sequence of characters so that the model best predicts output characters in the given sequence as described in Definition 4. This is called the *learning* or *training* phase for the model. The outputs of the training phase are simply ignored, the whole purpose of the process is to determine optimal parameters for predicting correct sequences. Then the model (represented by its parameters) is subsequently used to produce new output sequences when new input sequences are presented. This is called the *prediction* mode of the model. An RNN model capable of producing various input and output lengths of sequence-to-sequence predictions is presented in [27].

Both the *encoder* and *decoder* in Figure 4.3 perform the same function, which is represented schematically in Figure 4.4. The encoder/decoder function is performed as a nonlinear transformation between the input and output spaces. As before, we let S_J

denote the set of subsequences of length J in $\mathcal{R}$. The encoder/decoder function, denoted $RNN()$ and depicted in Figure 4.4, is the formal realization of a RNN model $M_{J,1}^{(26)}$, and it is recursively defined below.

Definition 7. [RNN function] The $RNN()$ function is defined as:

$$
\begin{aligned}
&RNN : S_J \rightarrow \overline{\mathcal{A}} \times (0,1) \\
&RNN(c_{i_1}, ..., c_{i_J}; H, W, Z, B_1, B_2) = \{(c_i, p_i)\} \\
&\text{where:} \\
&c_i = \overline{\mathcal{A}}[i],\ i = 1, ..., |\overline{\mathcal{A}}| \\
&[x_{i_1}, ..., x_{i_J}] = index([c_{i_1}, ..., c_{i_J}]) \\
&(p_1, ..., p_{|\overline{\mathcal{A}}|} = softmax(Wx_{i_J} + Zh_{i_{J-1}} + B_2) \\
&h_{i_{J-1}} = Relu(Wx_{i_{J-1}} + Zh_{i_{J-2}} + B_1) \\
&..... \\
&h_{i_1} = Relu(Wx_{i_1} + B_1)
\end{aligned}
\tag{4.2}
$$

That is, the $RNN()$ function in (4.2) and depicted in Figure 4.4 creates a pair character-probability (c_i, p_i) for each character in $\overline{\mathcal{A}}$. Overall, there are $|\overline{\mathcal{A}}| = 26$ pairs produced by the function on each input. The probability p_i of character $c_i = \overline{\mathcal{A}}[i]$ is computed using the $softmax()$ function on a nonlinear, recursive transformation of the numerical input $[x_{i_1}, ..., x_{i_J}]$, which is the list of indices for the sequence of characters $[c_{i_1}, ..., c_{i_J}] \in S_J$. Finally, the matrices W, Z, H, B_1, B_2 are the parameters of the $RNN()$ function, which are optimally determined to maximize the confidence of $RNN()$. Clearly, the $RNN()$ function in (4.2) describe a MOTRM model $M_{J,1}^{(26)}$ as per Definition 6.

We next establish how the parameters W, Z, H, B_1, B_2 of $RNN()$ are determined in practice. We first introduce the *cross entropy* $L(\cdot, \cdot)$ as a measure for RNN model confidence.

Algorithm 1 RNN Learning Phase (training a $M_{K,1}^{(26)}$ model)

procedure RNN LEARNING($\mathcal{R}, \overline{\mathcal{A}}, K, EPOCHS, BATCHSIZE$) ▷ computes the optimal RNN parameters
Input: $\mathcal{R}, \overline{\mathcal{A}}, K, EPOCHS$
Output: $W, Z, H, B_1, B_2, loss$
    $W, Z, H, B_1, B_2 \leftarrow$ $random\ values$
    for $k = 1, ..., EPOCHS$ **do**
        $loss \leftarrow 0$
        $batchcount \leftarrow 0$
        for $i = 1, ..., len(\mathcal{R}) - K - 1$ **do**
            $\{y_i \leftarrow$ argmax $(RNN([\mathcal{R}[i], ..., \mathcal{R}[i + K - 1]]; W, Z, H, B_1, B_2))$
            $loss = loss + L([\mathcal{R}[i], ..., \mathcal{R}[i + K - 1]])$
            $batchcount \leftarrow batchcount + 1$
            if $batchcount \mod BATCHSIZE = 0$ **then** ▷ optimize the loss function
                $W, Z, H, B_1, B_2 \leftarrow$ argmin $\left(\sum_{k=1}^{BATCHSIZE} L([\mathcal{R}[i + k], ..., \mathcal{R}[i + k + K - 1]]) \right)$
                $batchcount \leftarrow 0$
            end if
        end for
    end for
    return $W, Z, H, B_1, B_2, loss$ ▷ Returns the optimal parameters and estimated error
end procedure

Definition 8. [Cross-Entropy Loss] Given $RNN(c_{i_1}, ..., c_{i_J}; H, W, Z, B_1, B_2)$ the cross-entropy loss over a set of sequences S_J on input $[c_{i_1}, ..., c_{i_J}]$ is:

$$L([c_{i_1}, ..., c_{i_J}]) = -\log p_{i_{J+1}}$$

where $(c_{i_{J+1}}, p_{i_{J+1}}) \in RNN(c_{i_1}, ..., c_{i_J}; H, W, Z, B_1, B_2)$.

That is, the *cross-entropy loss* is the computed in terms of the probability of the character *following* the input sequence (which is the *expected* character), as computed by the $RNN()$ function.

I implement the *learning phase* of the sequence-to-sequence model described in Figure 4.3 in Algorithm 1. The algorithm computes the optimal values of the model parameters W, Z, H, b, which are subsequently conveyed to the model decoder for performing character prediction with high confidence. The algorithm takes as input the alphabet and the recovered manuscript text, together with some fine-tuning algorithm parameters: the number of epochs ($EPOCHS$) and the batch size ($BATCHSIZE$). It starts with random values of the model parameters W, Z, H, b (line 4), then iterates $EPOCHS$ times, scanning the whole recovered text at each iteration (lines 5-17). At each iteration, each subsequence of K characters is being analyzed (lines 8-16) by computing the predicted values through the $RNN()$ function with parameters W, Z, H, b, and the corresponding cross-entropy loss value (line 10). After each $BATCHSIZE$ group of iterations, the optimal values of parameters W, Z, H, b are being computed by minimizing the total cross-entropy loss for the group (line 13).

Both the RNN model $M_{K,1}^{(26)}$ in Definition 7 and the Algorithm 1 implementations in Python are given in Appendix A.5. An example of the results of a simple $M_{1,1}^{(26)}$ model is presented below.

Example 4.1. *On the $\mathcal{R}$ text in Appendix A.2 and alphabet* (4.1) *our RNN model imple-*

mentation $M_{1,1}^{(26)}$ *for one character input produces the following results:*

$$RNN([a]) = \{(\_, 0.3286), (5, 0.0063), (7, 0.0000), (a, 0.0000), (b, 0.0066), (c, 0.0402),$$

$$(d, 0.0158), (e, 0.0000), (f, 0.0062), (g, 0.0066), (h, 0.0233), (i, 0.0000), (l, 0.0708),$$

$$(m, 0.0312), (n, 0.2532), (o, 0.0000), (r, 0.0814), (s, 0.0303), (t, 0.0140), (u, 0.0062),$$

$$(w, 0.0142), (x, 0.0000), (y, 0.0000), (æ, 0.0000), (ð, 0.0413), (þ, 0.0237)\}$$

$$RNN([x]) = \{(\_, 0.0000), (5, 0.0000), (7, 0.0000), (a, 0.0000), (b, 0.0000), (c, 0.0000),$$

$$(d, 0.0000), (e, 0.0000), (f, 0.0000), (g, 0.0000), (h, 0.0000), (i, 0.0000), (l, 0.0000),$$

$$(m, 0.0000), (n, 0.0000), (o, 1.0000), (r, 0.0000), (s, 0.0000), (t, 0.0000), (u, 0.0000),$$

$$(w, 0.0000), (x, 0.0000), (y, 0.0000), (æ, 0.0000), (ð, 0.0000), (þ, 0.0000)\}$$

The model predicts various outputs from input a *(with output 'space' with the highest confidence of 32.86%), whereas on input* x *the model predicts output* o *with confidence 100%.*

4.3 The structural architecture the Hidden Markov Model

I establish next a Hidden Markov Model (HMM) that is specifically designed for performing manuscript text recovery. We will also show that there is an intrinsic connection between the HMM model we introduce here and the corresponding RNN model we defined in Section 4.2.

A typical HMM (described in Chapter 2) is depicted in Figure 4.5. For a HMM implementing the model $M_{K,1}^{(26)}$, the set of states X consists of all character in the alphabet (4.1), and the set of observations Y is the set S_K of all subsequences of length K in $\mathcal{R}$. In this model, based on an *observed* sequence of characters $[c_1, ..., c_K] \in S_K$, we aim to determine the subsequent character c_{K+1}, which is the *state* of the model in our approach.

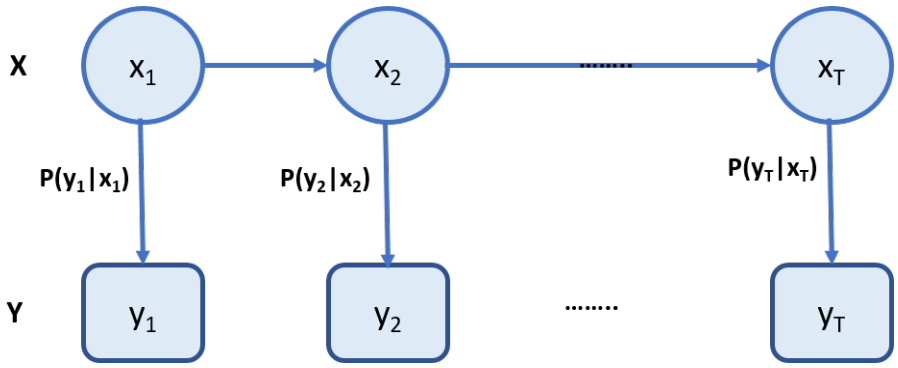


Figure 4.5: The HMM model over the sequence $\mathcal{T}$

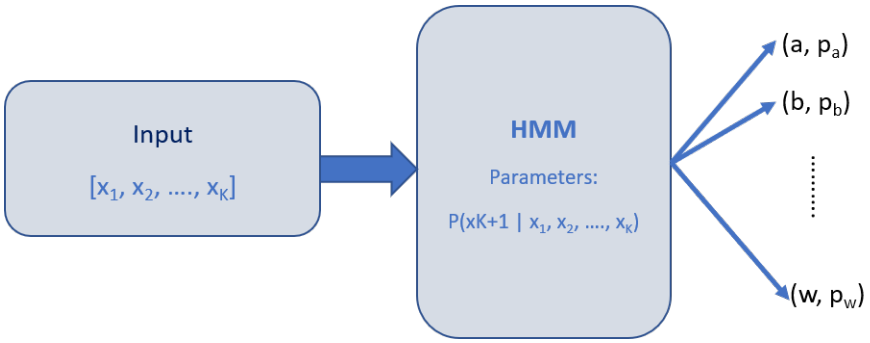


Figure 4.6: The HMM-based $M_{K,1}^{(26)}$

This HMM inference problem is called *filtering* and the goal is to determine the conditional probability $P(c_{K+1}|c_1, ..., c_K)$ (the probability of a character when all previous K characters are observed). We adopt the HMM filtering architecture to build the $M_{K,1}^{(26)}$ model for restoring missing manuscript text. Figure 4.6 illustrate this model. The model parameters are all conditional probabilities for one character when all previous K characters are observed, takes as input a sequence of characters $[x_1, ..., x_K] \in S_K$ of $\mathcal{R}$, and produces as output all pairs character-confidence for each character in the alphabet. The model defines a mapping, denoted as $HMM()$, which is formally defined as follows.

Definition 9. [HMM function] The $HMM()$ function over all S_J sequences in $\mathcal{R}$ is defined

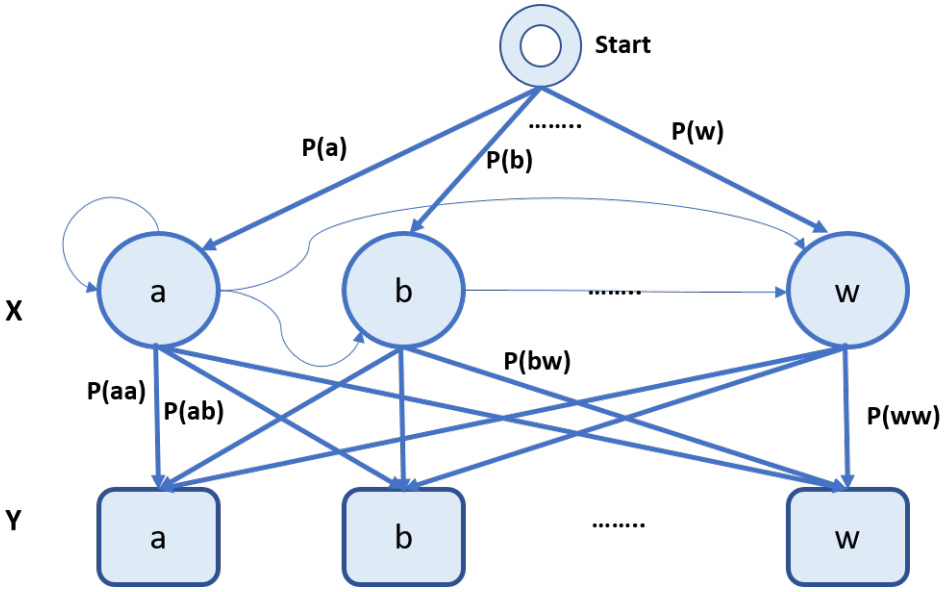


Figure 4.7: HMM model $M_{1,1}^{(26)}$

as:

$$
\begin{aligned}
&HMM : S_J \rightarrow \overline{\mathcal{A}} \times (0,1) \\
&HMM(c_{i_1}, ..., c_{i_J}) = \{(c_i, P(c_i|c_{i_1}, ..., c_{i_J}))|c_i \in \overline{\mathcal{A}}\} \\
&where : \\
&P(c_i|c_{i_1}, ..., c_{i_J}) = \frac{|\{[c_{i_1}, ..., c_{i_J}, c_i] \in S_{K+1}\}|}{|\{[c_{i_1}, ..., c_{i_J}] \in S_K\}|}
\end{aligned}
\tag{4.3}
$$

That is, the model parameters consists of *apriori* probabilities $P(c_i|c_{i_1}, ..., c_{i_J}) = \frac{|\{[c_{i_1}, ..., c_{i_J}, c_i] \in S_{K+1}\}|}{|\{[c_{i_1}, ..., c_{i_J}] \in S_K\}|}$, which are computed as the ratio of occurrences of sequence $[c_{i_1}, ..., c_{i_J}, c_i]$ over the number of occurrences of sequences $[c_{i_1}, ..., c_{i_J}]$, for all sequences $[c_{i_1}, ..., c_{i_J}] \in S_K$. Moreover, the confidence of predicting some character $c_i \in \overline{\mathcal{A}}$ on input the $[c_{i_1}, ..., c_{i_J}]$ is the respective conditional probability parameter $P(c_i|c_{i_1}, ..., c_{i_J})$.

Figure 4.7 shows an HMM-based model for $K = 1$. The parameters of the model consists of conditional probabilities for each sequence of two characters in $\mathcal{R}$. For simplicity we denote $P(aa) = P(a|a)$, $P(ab) = P(b|a)$, $P(bw) = P(w|b)$, etc. That is, the probability of character a when a is observed, of b when a is observed, of w when b is observed, etc.

These probabilities are computed as given in (4.3) and they are given in Appendix A.3 for the input text in Appendix A.2. An example of the results of a simple HMM-based $M_{1,1}^{(26)}$ model as in Figure 4.7 is presented below.

Example 4.2. *On the $\mathcal{R}$ text in Appendix A.2 and alphabet* (4.1) *our HNN model implementation $M_{1,1}^{(26)}$ for one character input produces the following results:*

$$HMM([a]) = \{(\_, 0.3145), (5, 0.0081), (7, 0.0000), (a, 0.0000), (b, 0.0081), (c, 0.0403),$$

$$(d, 0.0161), (e, 0.0000), (f, 0.0081), (g, 0.0081), (h, 0.0242), (i, 0.0000), (l, 0.0726),$$

$$(m, 0.0323), (n, 0.2500), (o, 0.0000), (r, 0.0806), (s, 0.0323), (t, 0.0161), (u, 0.0081),$$

$$(w, 0.0161), (x, 0.0000), (y, 0.0000), (æ, 0.0000), (ð, 0.0403), (þ, 0.0242)\}$$

$$HMM([x]) = \{(\_, 0.0000), (5, 0.0000), (7, 0.0000), (a, 0.0000), (b, 0.0000), (c, 0.0000),$$

$$(d, 0.0000), (e, 0.0000), (f, 0.0000), (g, 0.0000), (h, 0.0000), (i, 0.0000), (l, 0.0000),$$

$$(m, 0.0000), (n, 0.0000), (o, 1.0000), (r, 0.0000), (s, 0.0000), (t, 0.0000), (u, 0.0000),$$

$$(w, 0.0000), (x, 0.0000), (y, 0.0000), (æ, 0.0000), (ð, 0.0000), (þ, 0.0000)\}$$

The model predicts various outputs from input a *(with output 'space' with the highest confidence of 31.45%), whereas on input* x *the model predicts output* o *with confidence 100%. These results are very similar to the results in Example 4.1 for the corresponding RNN model. Notice that in an HMM model some confidence values are zero (as they are computed based on several occurrences for different sequences), whereas in an RNN model these confidences are never zero as they are computed via the softmax function in Definition* **??** *(but they might be very small and consequently approximated as zero as in Example 4.1).*

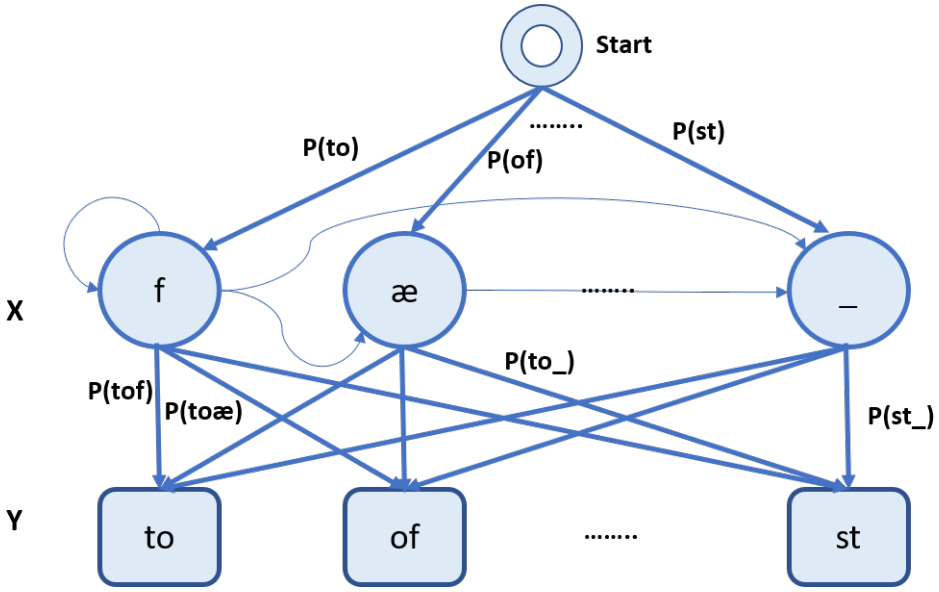


Figure 4.8: Transitions for the sequence 'tofæst '$\subseteq \mathcal{T}$

Figure 4.8 shows an HMM-based model for $K = 2$. That is, two characters are observed (they form the set of observations Y) and the prediction for the next character is computed, for each character in the alphabet (with their respective confidences), which form the set of HMM states. The model parameters consists of all conditional probabilities for sequences of length 3 in 'tofæst ', as follows: $P(tof) = P(f|to)$, $P(to\text{æ}) = P(\text{æ}|to)$, etc. As before, these probabilities also represent the confidences of predicting each respective alphabet character. The complexity of the HMM model (both in terms of architecture and computations) grows with K. Some numeric results from our experiments for $K = 2$ are presented in Chapter 5.

The RNN and HMM models as formulated in Definitions 7 and 9, the results presented in Examples 4.1 and 4.2, and the experimental results we present in Chapter 5 allow us formulate the following conclusion that establishes an intrinsic relationship between the RNN and HMM models.

A Recurrent Neural Network model for text recovery as formulated in Defini-

tion 7 *converges* to a Hidden Markov Model for text recovery as formulated in Definition 9.

CHAPTER 5

THE EXPERIMENTAL RESULTS

The models were implemented and experiments were run in Python. The Recurrent Neural Network (RNN) models were implemented using the TensorFlow [28] machine learning platform and the Keras [8] API. We used three folios (pages) of manuscript text for our experiments: the original manuscript text is listed in Appendix A, Section A.1, and the pre-processed text (after removing punctuation and meta-information) is listed in Section A.2. The complete code of Python experiments are listed in Appendix A, Section A.5.

I implemented both the RNN and HMM based versions of the general MOTRM model in Definition 6 for one output ($J = 1$) and the alphabet in (4.1), $M_{K,1}^{(26)}$. I subsequently ran experiments for $K = 1, 2, 3$, that is for the models $M_{1,1}^{(26)}$, $M_{2,1}^{(26)}$, and $M_{3,1}^{(26)}$. For all the experiments, I used two personal computers (PC) equipped with an Intel Core i7-4770 CPU @3.40GH and Intel Core i5 with 8GB RAM. The experiments took different time based on the input sequence and the output expectation with respect to the computational capabilities.

In this study, I implemented three model: model one takes one character input and predicts the next character. Model two take two character input and one character output, and model three takes three character input and one character output. All three models are trained and tested on a small portion from the beowulf manuscript. Model 1, which is a one to one model, trained and finished running on both the computers in under five minutes. Model two took one hour to train and finish running on Intel core i5 with 8GB RAM. On the Intern core i7 computer, model ran faster, although it was not an significant improvement. Finally model three, with an input sequence length of 3 took significantly longer time to train and test in both the computer. In the Intern i5 computer, with the same dataset as the other two models, this experiment ran over night. Similar timing was observed for Intel core i7, although it was faster.

It is worth mentioning that computational prowess is of utmost importance. These models are expensive to train, and is known to generate results with higher confidence for larger dataset. This brings the conundrum of training time and memory complexity. When I tried to train model two with the entire beowulf manuscript, the Intel core i5 computer ran out of memory. Considering the computational resources I decide to run experiments with limited data size, and smaller models. The implementation details and experimental results are reported in the subsequent sections.

5.1 THE RNN-BASED IMPLEMENTATION OF $M_{K,1}^{(26)}$

The model has been implemented in Python using the Keras libraries [8] and a “sequential" architecture.

```
def build_model(vocabulary, prediction_num,
                hidden_layers=256, n_fac=42):
    vocab_size = len(vocabulary)

    model = Sequential([
            SimpleRNN(units=hidden_layers,
                input_shape=(1,prediction_num),
                activation="relu"),
            Dense(vocab_size, activation='softmax')
        ])

    model.compile(loss='sparse_categorical_crossentropy',
            optimizer='adam')
    model.summary()
```

```
    return model
```

The model's parameters summary is listed below. The same architectural model has been used for all experiments ($K = 1, 2, 3$).

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 1, 8)              208
_________________________________________________________________
simple_rnn_5 (SimpleRNN)     (None, 128)               17536
_________________________________________________________________
dense_5 (Dense)              (None, 26)                3354
=================================================================
Total params: 21,098
Trainable params: 21,098
Non-trainable params: 0
```

For each experiment, the model has been trained for 1000 epochs with a batch size of 12. We have used a simple re-sampling technique (replicating the input data five times) to boost the model's accuracy and training time.

5.2 THE HMM-BASED IMPLEMENTATION OF $M_{K,1}^{(26)}$

The HMM model implementation of $M_{K,1}^{(26)}$ entirely relies on the definition for the conditional probability in Definition 9:

$$P(c_i|c_{i_1},...,c_{i_J}) = \frac{|\{[c_{i_1},...,c_{i_J},c_i] \in S_{K+1}\}|}{|\{[c_{i_1},...,c_{i_J}] \in S_K\}|}$$

The relevant code from listing in Appendix A.5 is given below:

```
def generate_counts(inp, vocab, data, c2i_map, i2c_map):
    my_preds = {}
    for c in vocab:
        my_preds[c] = 0

    arr = np.array([c2i_map[i] for i in inp])
    la = len(arr)
    #lazy search
    for i in range(len(data)-la):
        #check if arr found at this position
        found = True
        for j in range(la):
            if (arr[j] != data[i+j]):
                found = False
                break
        if (found):
            c = i2c_map[data[i+la]]
            my_preds[c] += 1


    return my_preds

def generate_probs(inp, vocab, data, c2i_map, i2c_map):
    counts = generate_counts(inp, vocab, data,
                char_to_index, index_to_char)
    s = sum(counts.values())
```

```
probs = counts.copy()
if (s > 0):
    for k,v in probs.items():
        probs[k] = v/s

return probs
```

The implementation essentially creates a table with a row entry for each sequence of K characters starting at every position in the input text $\mathcal{R}$. For each such entry, a count is kept for each alphabet character and for each occurrence of that character immediately following the K-length string being considered. These occurrences are then normalized per each row, to produce the desired conditional probability.

In summary, our HMM for the $M_{K,1}^{(26)}$ model on an input $\mathcal{R}$ with $len(\mathcal{R}) = N$ has $|\overline{\mathcal{A}}| = 26$ states and up to $N - K - 1$ possible observed values, one for each K-length substring of $\mathcal{R}$ plus one character for what follows to the K-length substring (this is an upper bound, as not all these substrings are distinct). Consequently there are $(N - K - 1) * (26 - 1)$ transition parameters (the only parameters of the HMM we are actually using) for our model[1]. For our input text $\mathcal{R}$ given in Appendix A.2, we have that $N = len(\mathcal{R}) = 2115$. Hence for $K = 1, 2, 3$ we have that the HMM model has a constant number of parameters 52,825; 52,800; 52,775, respectively.

5.3 CHARACTER PREDICTION USING THE $M_{1,1}^{(26)}$ MODELS

We ran the experiments repeatedly for different parameter values (epochs, batch sizes, re-sampling and no re-sampling). As expected, a large number of epochs and re-sampling considerably improved the results of our experiments. It was not our goal to study and re-

[1]The computation takes into account that one transition probability can be determined once the other are known, since their sum is one.

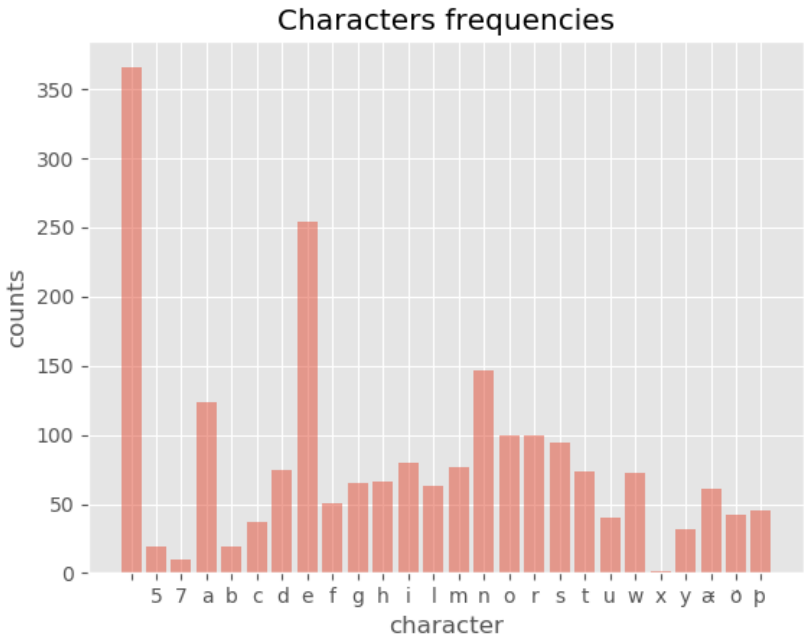


Figure 5.1: Characters frequencies

port the relationship computation power to accuracy. Our aim was to illustrate the intrinsic relation between the RNN and HMM models. We consider results for $K = 1$ we obtained using 1000 epochs, batch size 12 are exceptional in terms of justifying our claims.

We first computed all alphabet $\overline{\mathcal{A}}$ characters frequencies to determine how each character is used in the input text of our experiments. The results are illustrated in Figure 5.1. We see, for instance, that space is the most frequent (no surprise), followed by 'e', 'n', 'a', etc. We see that 'x' is less frequent, in fact, with zero occurrences. We then organized our experimental results as follows:

- Tabular versions of the $RNN()$ and $HMM()$ functions in Definitions 7 and 9.
- Next character predictions comparisons for the $RNN()$ and $HMM()$ functions.
- The descending order of probabilities/confidences of each alphabet character input.

The tabular versions of the $RNN()$ and $HMM()$ functions are completely given in Ap-

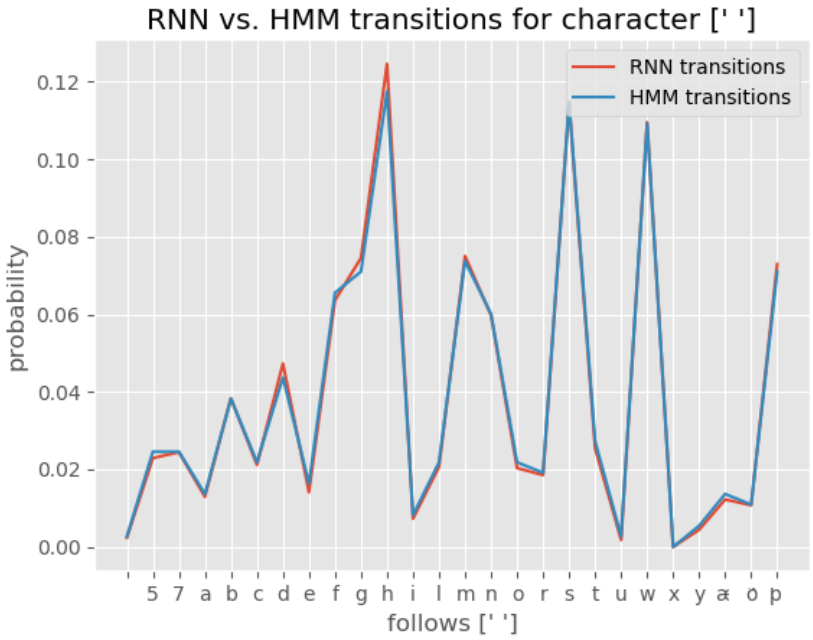


Figure 5.2: Probabilities of characters following the character ’ ’

pendix A.4 and A.3, respectively. The input character is given in the table heading, then the output probabilities/confidences are in the respective character column, for the corresponding character in the first column. For instance (as given also in Examples 4.1 and 4.1):

$$RNN([a]) = \{(\_, 0.3286), (5, 0.0063), (7, 0.0000), (a, 0.0000), (b, 0.0066), (c, 0.0402),$$

$$(d, 0.0158), (e, 0.0000), (f, 0.0062), (g, 0.0066), (h, 0.0233), (i, 0.0000), (l, 0.0708),$$

$$(m, 0.0312), (n, 0.2532), (o, 0.0000), (r, 0.0814), (s, 0.0303), (t, 0.0140), (u, 0.0062),$$

$$(w, 0.0142), (x, 0.0000), (y, 0.0000), (æ, 0.0000), (ð, 0.0413), (þ, 0.0237)\}$$

$$HMM([a]) = \{(\_, 0.3145), (5, 0.0081), (7, 0.0000), (a, 0.0000), (b, 0.0081), (c, 0.0403),$$

$$(d, 0.0161), (e, 0.0000), (f, 0.0081), (g, 0.0081), (h, 0.0242), (i, 0.0000), (l, 0.0726),$$

$$(m, 0.0323), (n, 0.2500), (o, 0.0000), (r, 0.0806), (s, 0.0323), (t, 0.0161), (u, 0.0081),$$

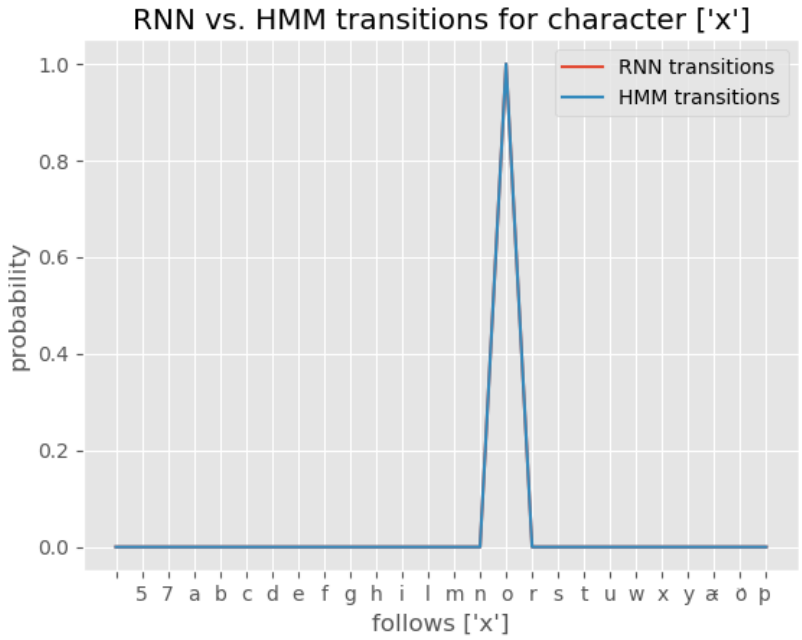


Figure 5.3: Probabilities of characters following 'x'

$(w, 0.0161), (x, 0.0000), (y, 0.0000), (æ, 0.0000), (ð, 0.0403), (þ, 0.0242)\}$

Comparisons of various characters probabilities/confidences can be seen in Figures 5.2, 5.3, and 5.5 for space character (highest frequency), 'x' (least frequency), 'g' (intermediate frequency, respectively. Also, 5.6 shows the probabilities of characters following the special character '5' (placeholder for 'missing'), as potentially useful for predicting the missing part. In these figures, the probabilities are indicated on the 'y' axis and the next characters on the 'x' axis. The fact that the graphs for $RNN()$ and $HMM()$ are almost perfectly overlapping suggest the two models behave similarly. However, as pointed out earlier, these experimental results depend greatly on the parameters of the model. For comparison, Figure 5.4 shows the transitions to the next character from the least frequent character 'x', for training with 25 epochs and no re-sampling the data (instead of training with 1000 epochs as for the results in 5.3). The differences between the two curves are significant.

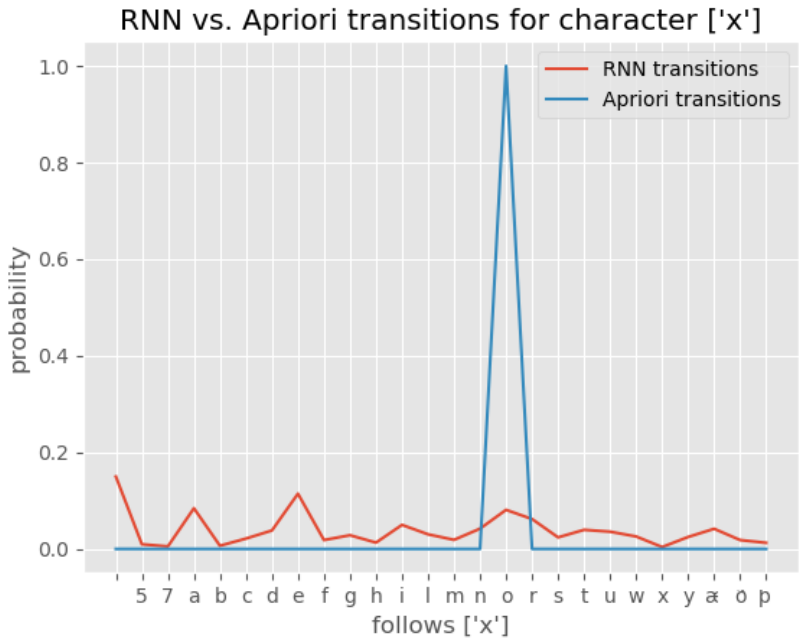


Figure 5.4: Probabilities of characters following 'x': lesser trained model

Finally, the descending order of probabilities/confidences of each alphabet character input is fully given in Tables 5.1 and 5.2. For instance, the first row of Table 5.1 shows that $RNN(' ')$ predicts the following characters in the descending confidence order: 'hswmgþfndbt75cloreaæðiy ux'. On the other hand, on the same input 'space', the $HMM(' ')$ model predicts: 'hswmgþfndbt75cloreaæðiyu x'. We can see a perfect match on all but the last two before las places. We consider these results as excellent since, in practice, only the first few options are being considered. A quick investigation of Tables 5.1 and 5.2 shows no difference on the first position for the two models. However, some input do produce different outcomes starting at the second position (inputs 'b', 'c', etc.). The following example shows both functions $RNN()$ and $HMM()$ at work: prediction of missing characters in $\mathcal{R}$.

Example 5.1. *Lines 3-4 in the models' input text (Appendix A.2) read:*

fæhðe 7fyrene wæs tofæst onþam þ5

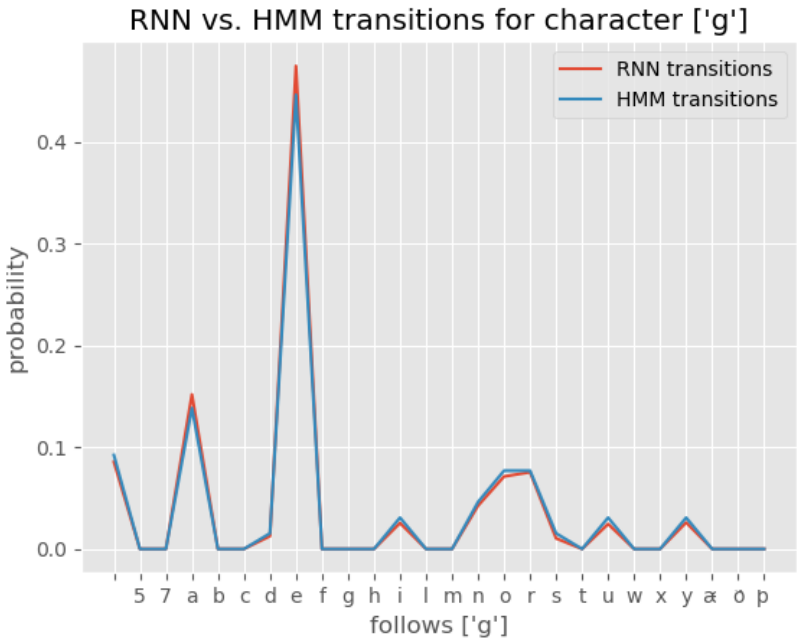


Figure 5.5: Probabilities of characters following the 'g'

wæs eað fynde þe him elles hwær geru5

To predict the missing characters at the end of each line we use $RNN('þ')$, $HMM('þ')$, $RNN('u')$, *and* $HMM('u')$ *and we retain the first five character predictions following the respective inputs, to get, respectively:*

$RNN('þ') \rightarrow$ *'eæoag',*

$HMM('þ') \rightarrow$ *'eæoam',*

$RNN('u') \rightarrow$ *'mnþrð', and*

$HMM('u') \rightarrow$ *'mnrþ '*

Now if we compare with the respective lines in $\mathcal{T}$ *from [18]:*

fæhðe 7fyrene wæs tofæst onþam þ(a)

wæs eað fynde þe him elles hwær geru(m)

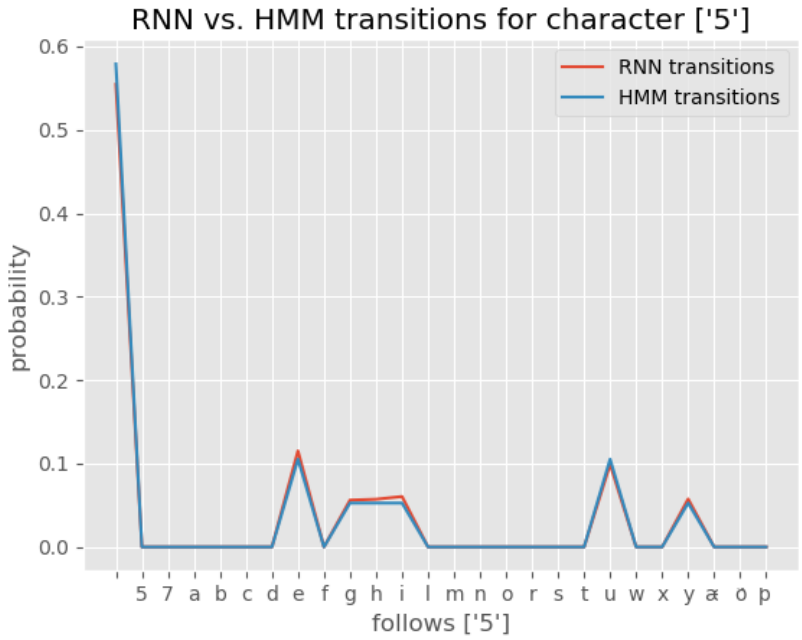


Figure 5.6: Probabilities of characters following '5': unknown character

We conclude that the first prediction matched the expert's opinion ('a') only on the 4th position for both models, whereas the second prediction was correct ('m') on the first position for both models.

The example above emphasizes once again that these models are meant to assist an expert, rather than replacing the work of the expert.

5.4 Character prediction using the $M_{2,1}^{(26)}$ models

For $K = 2$ the observable values consist of all (distinct) substrings of 2 characters in the input text. We collected the frequencies for all such substrings and the top 20 most frequent substrings are shown in Figure 5.7.

However, the slightly increased complexity of the $M_{2,1}^{(26)}$ model makes training such models more difficult. This is illustrated by the differences between transition probabilities of the RNN and HMM models on the input string 'ne' with relatively high frequency

Model / Character	Next character(s)
’[RNN] ’ ’[HMM] ’	’hswmgþfndbt75cloreaæðiy ux’ ’hswmgþfndbt75cloreaæðiyu x’
’[RNN] 5’ ’[HMM] 5’	’ euiyhgrdtnlðæbsfmþacwo7x5’ ’ euyghioðæxwtsrmn5lfdcba7þ’
’[RNN] 7’ ’[HMM] 7’	’fwetlgsn dmcrhðþibau5æyox7’ ’fwegtlsn rðæyxumo5ihdcba7þ’
’[RNN] a’ ’[HMM] a’	’ nrlðcmsþhdwtgb5fueyiæx7ao’ ’ nrlðcsmþhwtdbugf5oeixyæ7a’
’[RNN] b’ ’[HMM] b’	’eioaæbru dchyntwsfðmþlx57g’ ’eoiabæur ðyxwtsmn5lhgfdc7þ’
’[RNN] c’ ’[HMM] c’	’e yunowrð5tiæamþgsflhdcb7x’ ’eyu noðwr5æxtsmlihgfdcba7þ’
’[RNN] d’ ’[HMM] d’	’e araæoudniywlðtþmhgf57bxsc’ ’e araæouydwilnðstxm5hgfcb7þ’
’[RNN] e’ ’[HMM] e’	’ anoslrtmdwgcfhþbðeu5iæ7xy’ ’ anoslrtmdwgfchþðbe57uixyæ’
’[RNN] f’ ’[HMM] f’	’eærtyoa du5ilwðmhngsþfcb7x’ ’eæryto adu5ðxwsmnlihgfcb7þ’
’[RNN] g’ ’[HMM] g’	’ea ronyiudslæmðtxhgb75fcwþ’ ’ea ronyuidsðæxwtm5lhgfcb7þ’
’[RNN] h’ ’[HMM] h’	’etiwðy ærua5odnlmghsþfcb7x’ ’etiwðy æura5oxsmnlhgfdcb7þ’
’[RNN] i’ ’[HMM] i’	’ðnhmgdcslaefþ5btx wru7oiyæ’ ’ðnhmgscdlaeþf5btxioruwyæ7 ’
’[RNN] l’ ’[HMM] l’	’ eadiolcmurnwyfætxgbð7sþh5’ ’ eadiolumcwfnrygðæxs7thb5þ’

Table 5.1: Part I: Transitions to next character for RNN and HMM

Model / Character	Next character(s)
’[RNN] m’ ’[HMM] m’	’ oaeiæuy5g7rmhwnfðxþltcdsb’ ’ oaeiæuy5ðxwtsrmnlhgfdcb7þ’
’[RNN] n’ ’[HMM] n’	’ ediagocnðumþftæyrsxlhb5w7’ ’ ediagocnðuæytrmfþ5slhwxb7’
’[RNN] o’ ’[HMM] o’	’nrdlf tmðgþhbcs5euiyoæaw7x’ ’nrdlf tmðgþ5bschoai7uwxyæe’
’[RNN] r’ ’[HMM] r’	’e aiuæhndðofgymrcwþlts5x7b’ ’e aiuæhnofdðgswytmrlcþ5xb7’
’[RNN] s’ ’[HMM] s’	’ etwicsæanoylufgmþhb7rðd5x’ ’ tewcisæaynouldxmr5hgfdb7þ’
’[RNN] t’ ’[HMM] t’	’ eoruaigtwfæbxmþh7dcnslð5y’ ’ eoruaifgwtðæyxsmn5lhdcb7þ’
’[RNN] u’ ’[HMM] u’	’mnþrð gsfl5hdwte7boiæacyux’ ’mnrþ ðsfgl5æyxwutb7caihedo’
’[RNN] w’ ’[HMM] w’	’iæaeoyrlu tdnsþfcðwmhg5x7b’ ’iæeaoyr uldxwtsmn5hgfdcb7þ’
’[RNN] x’ ’[HMM] x’	’o 5ðæyxwutsrnmlihgfedcba7þ’ ’o 5ðæyxwutsrnmlihgfedcba7þ’
’[RNN] y’ ’[HMM] y’	’rnldðmthsoæiy7aexfþgw5cbu ’ ’rnlðdmthsoæyxwu 5igfecba7þ’
’[RNN] æ’ ’[HMM] æ’	’stgrdmþhclfnðb7x5w ouyæiae’ ’stgrdmþchlfðneuæyxwb7ao5i ’
’[RNN] ð’ ’[HMM] ð’	’ aeoluþnfwræi7ytsðdbmxcgh5’ ’ aeuolfnþgtðæyxw7shrbc5dim’
’[RNN] þ’ ’[HMM] þ’	’eæoagmuð75 yrindfltsþchxbw’ ’eæoamug5ðyr i7lnhfstdwxcbþ’

Table 5.2: Part II: Transitions to next character for RNN and HMM

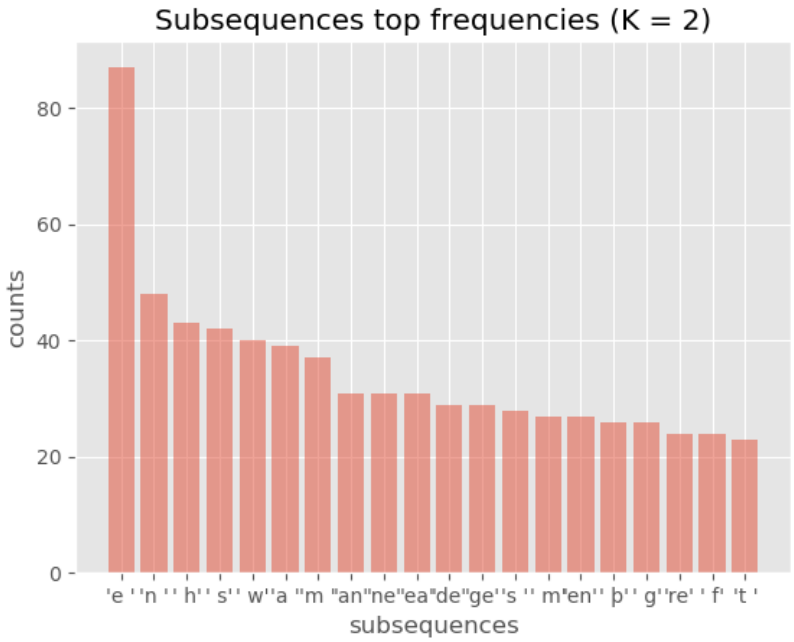


Figure 5.7: Most frequent substrings for the $M_{2,1}^{(26)}$ models

(Figure 5.8). For the relatively less frequent input string 'ru', the differences between the two models grow bigger (Figure 5.9). The following example will show that both models can still produce good practical results when predicting the next character.

Example 5.2. *We consider again line 4 in the models' input text (Appendix A.2):*

wæs eað fynde þe him elles hwær geru5

Notice that we omit line 3 this time, as in line 3 two preceding characters for the placeholder '5' include a space, so we omit that case from a practical perspective. To predict the missing characters at the end the line we use $RNN('ru')$*, and* $HMM('ru')$ *and, as before, we retain the first five character predictions following the respective inputs, to get, respectively:*

$RNN('ru') \rightarrow$ *'m eay', and*

$HMM('ru') \rightarrow$ *'mn 57'*

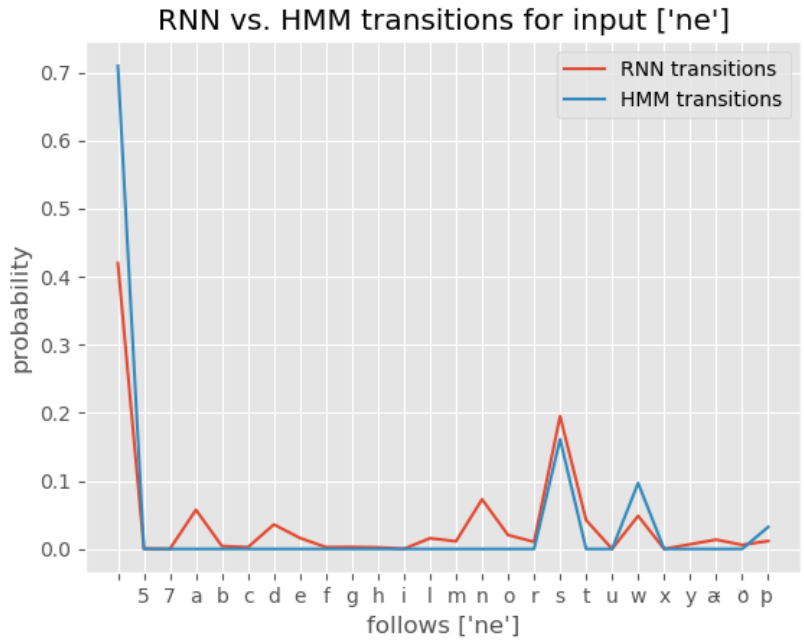


Figure 5.8: Next character probability for input 'ne' in $M_{2,1}^{(26)}$ model

Now if we compare with the respective line in $\mathcal{T}$ from [18]:

wæs eað fynde þe him elles hwær geru(m)

We conclude that the prediction was correct ('m') on the first position for both models.

5.5 Character prediction using the $M_{3,1}^{(26)}$ models

The observable values now consist of all (distinct) substrings of 3 characters in the input text. We collected the frequencies for all such substrings and the top 10 most frequent substrings are shown in the following figure. It is noticeable that, as expected, the frequencies of substrings went down considerably with the length of the input substring.

As we noticed in the previous section, the increased complexity of the $M_{3,1}^{(26)}$ model makes training these models more computationally expensive if good predictions are expected. This is illustrated by the differences between transition probabilities of the RNN

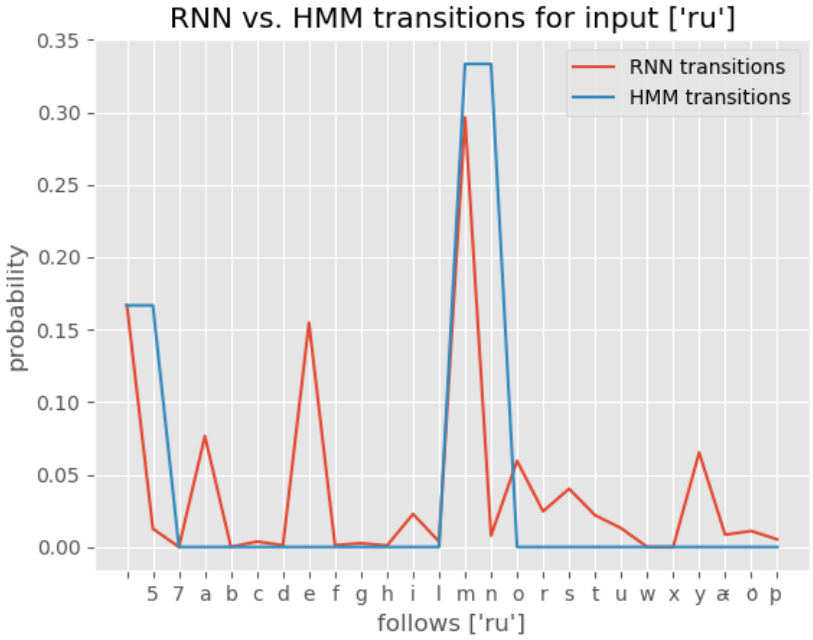


Figure 5.9: Next character probability for input 'ru' in $M_{2,1}^{(26)}$ model

and HMM models on the input string 'iht' with relatively high frequency (Figure 5.11). For our case scenario used in previous examples (this time a substring of length $K = 3$) 'eru' (for which we found frequency one), the differences between the two models grow bigger (Figure **??**). However, as the next example will illustrate an interesting outcome for this case scenario: the HMM model has not previous knowledge to make a prediction for the following character (other than the missing text placeholder '5'), whereas the RNN model does make a correct prediction.

Example 5.3. *As in the previous example, we consider again line 4 in the models' input text (Appendix A.2):*

wæs eað fynde þe him elles hwær geru5

This time, to predict the missing characters at the end the line we use the previous three characters as input $RNN('eru')$, and $HMM('eru')$ and, as before, we retain the first five

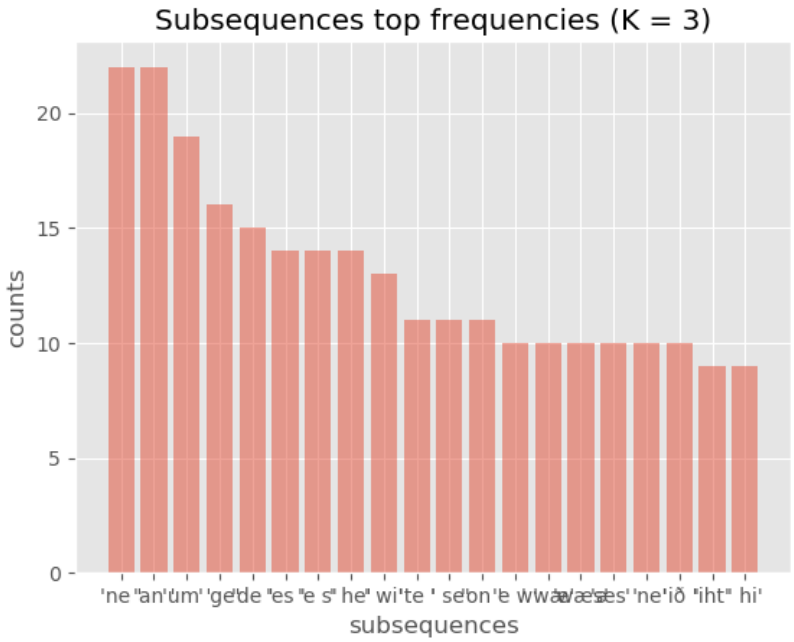


Figure 5.10: Most frequent substrings for the $M_{3,1}^{(26)}$ models

character predictions following the respective inputs, to get, respectively:

$RNN('eru') \rightarrow$ *'mr 5t', and*
$HMM('eru') \rightarrow$ *'5'*

Now if we compare again with the respective line in $\mathcal{T}$ from [18]:

wæs eað fynde þe him elles hwær geru(m)

An interesting thing happens. We conclude that the RNN prediction was correct ('m') on the first position, but the HMM model predicts '5' with 100% confidence. This is a direct consequence of the fact that HMM's prediction strictly relies on previously seen *similar instances of the input. As there is a single 'eru' subsequence in the whole input, the next character of this subsequence is indicated as the probable outcome with 100% confidence. The RNN model, on the other hand, is more flexible for this situations. Not*

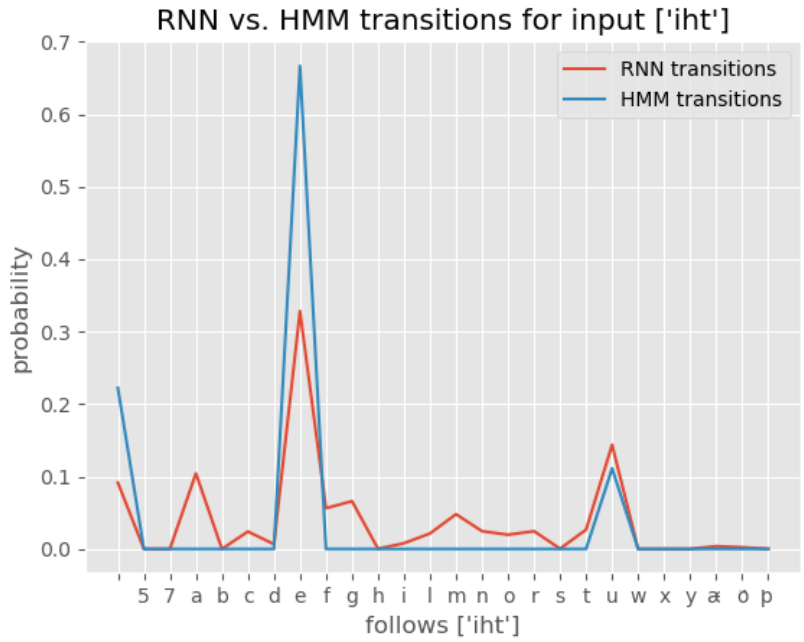


Figure 5.11: Next character probability for input'iht' in the $M_{3,1}^{(26)}$ model

only the RNN model relies on the previously seen 'eru' sequences, but also on the shorter 'ru' and 'u' (shorter length) sequences. This clearly shows the flexibility of the RNN models over the HMM counterparts.

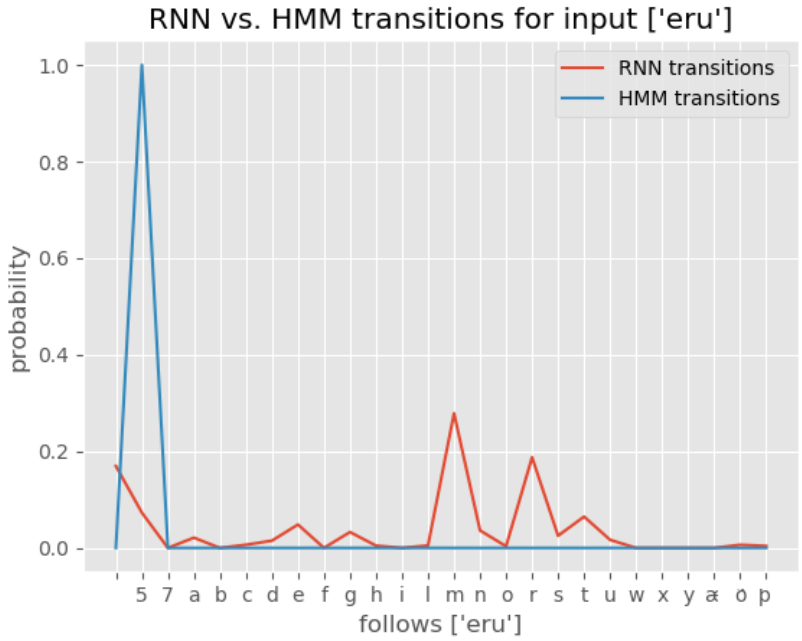


Figure 5.12: Next character probability for input 'eru' in the $M_{3,1}^{(26)}$ model

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this work we present a Recurrent Neural Network (RNN) model for finding missing character sequences in an Old English manuscript. Our model relies on the intrinsic relationship between Recurrent Neural Networks models and Hidden Markov Models (HMM), which we explain and formalize. While RNNs are very powerful models used nowadays for various tasks in deep learning (sequence classification, speech to text and text to speech conversions, etc.), they are difficult to explain and analyze from a theoretical standpoint. Moreover, due to the complexity of the RNN models, they may be unsuitable for some practical applications (such as online applications). The HMMs, on the other hand, are typically faster and easier to use than the RNNs counterparts. Their simplicity and theoretically sound foundation recommend them as well for various prediction tasks (one famous application is the Google's PageRank algorithm [20] used for search results ranking) and make HMMs suitable for character sequence prediction using significantly less computational power than their RNNs counterparts.

We formally define in Chapter 4 two models for text recovery: the Recurrent Neural Network (RNN) model and the Hidden Markov Mode (HMM). We also establish a remarkable relationship between the two models, namely that the RNN model converges to a HMM model. While the conclusion is very important, it lacks some practical aspects such as establishing bounds for such convergence. We leave this aspects, together with a formal justification for the conclusion for future work.

The analysis we carry in this study is performed at the character level. That is, given a sequence of characters of length K our model predicts the preceding or following sequence of characters of length L. Our specific goal is rather practical. We aim to assist restoration experts in their manuscript transcription work by providing quantitative prediction on missing parts of the manuscript and various options for manuscript restoration. As we shown

in Sections 5.1 and 5.2, the HMM model is considerable bigger (twice the number of parameters). However, as the complexity of the problem increases (longer input strings), the RNN model needs to adjust and increase its size in order to perform better. It would consequently need more network layers and more parameters, eventually catching up with the HMM model (which size will remain about the same). As illustrated by our experimental results, as the complexity of the problem increases, the RNN model requires either more training or more parameters in order to maintain its accuracy. It would be interesting to study in what theoretical conditions the two models will break even.

The character level approach we develop in this study can be extended or completed with an analogous word level approach: given a sequence of k words, the sequence of preceding/following L words is to be predicted. Our proof of concept character level approach was a practical choice dictated rather by practical reasons. We implemented and performed experiments on a small fragment of the manuscript text using relatively small models and a regular computer. A word level approach would require processing more data, using larger models and significantly more computational power. We leave this study for future work.

While our focus was on studying a single Old English manuscript (the Electronic Beowulf [18], for rather practical reasons), the results of this work can be directly extended, in most cases, to analyze and restore other manuscripts. Moreover, a complete analysis of resources and the computation power needed to analyze the *entire manuscript* can be carried out in detail. Such an analysis would provide a complete insight of how RNN and HMM models fare in a practical setting. This task was beyond the aims of our work and we consequently leave it for future work.

REFERENCES

[1] Matthew J. Beal, Zoubin Ghahramani, and Carl E. Rasmussen, *The infinite hidden Markov model*, Advances in Neural Information Processing Systems, 2002, pp. 577–584.

[2] Y. Bengio and P. Frasconi, *Input-output HMMs for sequence processing*, IEEE Transactions on Neural Networks **7** (1996), no. 5, 1231–1249.

[3] Yoshua Bengio, *Learning Deep Architectures for AI*, Found. Trends Mach. Learn. **2** (2009), no. 1, 1–127.

[4] Sam Blasiak and Huzefa Rangwala, *A hidden markov model variant for sequence classification*, Twenty-Second International Joint Conference on Artificial Intelligence, 2011.

[5] Björn Bringmann, Siegfried Nijssen, and Albrecht Zimmermann, *Pattern-Based Classification: A Unifying Perspective*, arXiv:1111.6191 [cs] (2011).

[6] Luigi Cerulo, Michele Ceccarelli, Massimiliano Di Penta, and Gerardo Canfora, *A Hidden Markov Model to detect coded information islands in free text*, 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM) (Eindhoven, Netherlands), IEEE, September 2013, pp. 157–166 (en).

[7] Gang Chen, *A Gentle Tutorial of Recurrent Neural Network with Error Backpropagation*, arXiv:1610.02583 [cs] (2018) (en).

[8] François Chollet et al., *Keras*, `https://github.com/fchollet/keras`, 2015.

[9] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio, *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*, arXiv:1412.3555 [cs] (2014).

[10] Sean R Eddy, *What is a hidden Markov model?*, Nature Biotechnology **22** (2004), no. 10, 1315–1316 (en).

[11] Philippe Fournier-Viger, Jerry Chun-Wei Lin, Rage Uday Kiran, Yun Sing Koh, and Rincy Thomas, *A survey of sequential pattern mining*, Data Science and Pattern Recognition **1** (2017), no. 1, 54–77.

[12] Alex Graves, *Generating Sequences With Recurrent Neural Networks*, arXiv:1308.0850 [cs] (2014).

[13] Alex Graves and Jürgen Schmidhuber, *Offline handwriting recognition with multi-dimensional recurrent neural networks*, Advances in Neural Information Processing Systems, 2009, pp. 545–552.

[14] Michiel Hermans and Benjamin Schrauwen, *Training and analysing deep recurrent neural networks*, Advances in Neural Information Processing Systems, 2013, pp. 190–198.

[15] Sepp Hochreiter and Jürgen Schmidhuber, *Long Short-Term Memory*, Neural Computation **9** (1997), no. 8, 1735–1780.

[16] David F. Johnson, *Widsið*, pp. 1–3, American Cancer Society, 2017.

[17] Kevin Kiernan, *Alternating currents in the electrifying classroom*, Teaching Beowulf in the Twenty-first Century (Tempe, AZ), vol. Medieval and Renaissance Texts and Studies, Arizona State University, 2014, pp. 55–66.

[18] ———, *Electronic Beowulf*, https://ebeowulf.uky.edu, last retrieved: February 2020.

[19] Tomas Mikolov, Quoc V. Le, and Ilya Sutskever, *Exploiting Similarities among Languages for Machine Translation*, arXiv:1309.4168 [cs] (2013).

[20] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd, *The pagerank citation ranking: Bringing order to the web.*, WWW 1999, 1999.

[21] Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio, *How to Construct Deep Recurrent Neural Networks*, arXiv:1312.6026 [cs, stat] (2014) (en).

[22] Thai-Hoang Pham and Phuong Le-Hong, *End-to-end recurrent neural network models for vietnamese named entity recognition: Word-level vs. character-level*, International Conference of the Pacific Association for Computational Linguistics, Springer, 2017, pp. 219–232.

[23] L. Rabiner and B. Juang, *An introduction to hidden Markov models*, IEEE ASSP Magazine **3** (1986), no. 1, 4–16.

[24] David E Rumelhart, *Parallel distributed processing: Explorations in the microstructure of cognition*, Learning internal representations by error propagation **1** (1986), 318–362.

[25] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams, *Learning representations by back-propagating errors*, nature **323** (1986), no. 6088, 533–536.

[26] Ilya Sutskever, James Martens, and Geoffrey E. Hinton, *Generating text with recurrent neural networks*, Proceedings of the 28th International Conference on Machine Learning (ICML-11), 2011, pp. 1017–1024.

[27] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le, *Sequence to sequence learning with neural networks*, 2014.

[28] TensorFlow Official Website, *An end-to-end open source machine learning platform*, https://www.tensorflow.org/, February 27, 2020.

[29] P.J. Werbos, *Backpropagation through time: What it does and how to do it*, Proceedings of the IEEE **78** (1990), no. 10, 1550–1560.

[30] T. Wessels and C.W. Omlin, *Refining hidden Markov models with recurrent neural networks*, Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium, vol. 2, July 2000, pp. 271–276 vol.2.

[31] Frank Wood, Cédric Archambeau, Jan Gasthaus, Lancelot James, and Yee Whye Teh, *A stochastic memoizer for sequence data*, Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09 (Montreal, Quebec, Canada), ACM Press, 2009, pp. 1–8 (en).

[32] J.P. Yamron, I. Carp, L. Gillick, S. Lowe, and P. van Mulbregt, *A hidden Markov model approach to text segmentation and event tracking*, Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181), vol. 1, May 1998, pp. 333–336 vol.1.

[33] Kwan Yi and Jamshid Beheshti, *A hidden Markov model-based text classification of medical documents*, Journal of Information Science **35** (2009), no. 1, 67–81.

[34] Ning Zhong, Yuefeng Li, and Sheng-Tang Wu, *Effective Pattern Discovery for Text Mining*, IEEE Transactions on Knowledge and Data Engineering **24** (2012), no. 1, 30–44.

Appendix A

APPENDIX A

A.1 ORIGINAL MANUSCRIPT TEXT FOR ANALYSIS

...)yrst acym(...) ane niht ef(...
morð beala mare 7nomearn fore
fæhðe 7fyrene wæs tofæst onþam þ(a)
wæs eað fynde þe him elles hwær geru(m)
licor ræste bed æfter burum ðahi(m)
gebeacnod wæs gesægd soðlice sweo(...
lan tacne heal ðegnes hete heold h(...
ne syðþan fyr 7fæstor seþæm feonde
æt wand. Swa rixode 7wið rihte wan
ana wið eallum oð þæt idel stod husa selest
wæs seo hwil micel .xii. wintra tid torn ge
þolode wine scyldenda weana gehwelcne
sidra sorga forðam wearð ylda bearnum
undyrne cuð gyddum geomore þætte gren
del wan hwile wið hroþgar hete niðas
wæg fyrene 7fæhðe fela missera singa
le sæce sibbe newolde wið manna hwone
mægenes deniga feorh bealo feorran
fea þingian neþær nænig witena wenan
þorfte beorhtre bote tobanum folmum .
f. 133v, (ll. 159–181)
...h)tende wæs deorc deaþ sc(...

...)uguþe 7geogoþe seomade 7syrede
...)in nihte heold mistige moras men ne
...)un non hwyder helrunan hwyrftum
scriþað swa fela fyrena feond mancyn
...)es atol angengea oft gefremede.
(h)eardra hynða heorot eardode sinc
...)ge sel sweartum nihtum nohe þone
gif stol gretan moste maþðum formeto
de ne his myne wisse þætwæs wræc micel
wine scyl dinga modes brecða monig oft
gesæt rice to rune ræd eahtedon hwæt
swið ferhðum selest wære wið fær gryrum
toge frem manne. Hwilum hie gehe
ton æt hrærg trafum wig weorþunga
wordum bædon þæt him gast bona geoce
gefremede. wið þeod þreaum swylc wæs
þeaw hyra. hæþenra hyht helle gemun
don inmod sefan metod hie ne cuþon
dæda demend newiston hie drihten god.
f. 134r, (ll. 182–203a)
...) huru heofena helm herian (...
cuþon wuldres waldend wabið þæm ðe
sceal þurh sliðne nið sawle bescufa(...
infyres fæþm frofre newenan wihte ge
wendan wel bið þæm þemot æfter deað
dæge drihten secean. 7tofæder fæþmum

freoðo wilnian.

.III.

Swaða mæl ceare maga healfdenes singa
la seað ne mihte snotor hæleð wean on
wendan wæs þætgewin to swyð laþ7longsum þe
onðaleode becom nydwracu niþgrim niht
bealwa mæst þætfram ham ge frægn higela
ces þegn god mid geatum grendles dæda
sewæs moncynnes mægenes strengest on
þæm dæge þysses lifes æþele 7eacen het
him yðlidan godne gegyrwan cwæð heguð
cyning ofer swan rade secean wol de mær
ne þeoden þahim wæs man na þearf ðone
siðfæt him snotere ceorlas lyt hwon logon.

A.2 MODEL TRAINING TEXT

5yrst acym5 ane niht ef5

morg beala mare 7nomearn fore

fæhge 7fyrene wæs tofæst onsam s5

wæs eag fynde se him elles hwær geru5

licor ræste bed æfter burum gahi5

gebeacnod wæs gesægd soglice sweo5

lan tacne heal gegnes hete heold h5

ne sygsan fyr 7fæstor sesæm feonde

æt wand. Swa rixode 7wig rihte wan

ana wig eallum og sæt idel stod husa selest

wæs seo hwil micel wintra tid torn ge

solode wine scyldenda weana gehwelcne

sidra sorga forgam wearg ylda bearnum

undyrne cug gyddum geomore sætte gren

del wan hwile wig hrosgar hete nigas

wæg fyrene 7fæhge fela missera singa

le sæce sibbe newolde wig manna hwone

mægenes deniga feorh bealo feorran

fea singian nesær nænig witena wenan

sorfte beorhtre bote tobanum folmum

5tende wæs deorc deas sc5

5uguse 7geogose seomade 7syrede

5in nihte heold mistige moras men ne

5un non hwyder helrunan hwyrftum

scrisag swa fela fyrena feond mancyn

5es atol angengea oft gefremede.
5eardra hynga heorot eardode sinc
5ge sel sweartum nihtum nohe sone
gif stol gretan moste masgum formeto
de ne his myne wisse sætwæs wræc micel
wine scyl dinga modes brecga monig oft
gesæt rice to rune ræd eahtedon hwæt
swig ferhgum selest wære wig fær gryrum
toge frem manne. Hwilum hie gehe
ton æt hrærg trafum wig weorsunga
wordum bædon sæt him gast bona geoce
gefremede. wig seod sreaum swylc wæs
seaw hyra. hæsenra hyht helle gemun
don inmod sefan metod hie ne cuson
dæda demend newiston hie drihten god.
5 huru heofena helm herian 5
cuson wuldres waldend wabig sæm ge
sceal surh sligne nig sawle bescufa5
infyres fæsm frofre newenan wihte ge
wendan wel big sæm semot æfter deag
dæge drihten secean. 7tofæder fæsmum
freogo wilnian.
Swaga mæl ceare maga healfdenes singa
la seag ne mihte snotor hæleg wean on
wendan wæs sætgewin to swyg las7longsum se
ongaleode becom nydwracu nisgrim niht

bealwa mæst sætfram ham ge frægn higela
ces segn god mid geatum grendles dæda
sewæs moncynnes mægenes strengest on
sæm dæge sysses lifes æsele 7eacen het
him yglidan godne gegyrwan cwæg hegug
cyning ofer swan rade secean wol de mær
ne seoden sahim wæs man na searf gone
sigfæt him snotere ceorlas lyt hwon logon.

A.3 Apriori (HMM) characters transition matrix

	5	7	a	b	c	d	e	f	
	0.0027	0.5789	0.0000	0.3145	0.0000	0.1081	0.2400	0.3425	0.0392
5	0.0246	0.0000	0.0000	0.0081	0.0000	0.0270	0.0000	0.0000	0.0196
7	0.0246	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
a	0.0137	0.0000	0.0000	0.0000	0.0526	0.0000	0.0933	0.1220	0.0392
b	0.0383	0.0000	0.0000	0.0081	0.0526	0.0000	0.0000	0.0039	0.0000
c	0.0219	0.0000	0.0000	0.0403	0.0000	0.0000	0.0000	0.0157	0.0000
d	0.0437	0.0000	0.0000	0.0161	0.0000	0.0000	0.0133	0.0197	0.0196
e	0.0164	0.1053	0.1000	0.0000	0.5263	0.3514	0.3867	0.0000	0.2157
f	0.0656	0.0000	0.3000	0.0081	0.0000	0.0000	0.0000	0.0157	0.0000
g	0.0710	0.0526	0.1000	0.0081	0.0000	0.0000	0.0000	0.0157	0.0000
h	0.1175	0.0526	0.0000	0.0242	0.0000	0.0000	0.0000	0.0079	0.0000
i	0.0082	0.0526	0.0000	0.0000	0.1053	0.0000	0.0133	0.0000	0.0000
l	0.0219	0.0000	0.1000	0.0726	0.0000	0.0000	0.0133	0.0669	0.0000
m	0.0738	0.0000	0.0000	0.0323	0.0000	0.0000	0.0000	0.0236	0.0000
n	0.0601	0.0000	0.1000	0.2500	0.0000	0.0811	0.0133	0.1063	0.0000
o	0.0219	0.0000	0.0000	0.0000	0.1053	0.0541	0.0533	0.0866	0.0784
r	0.0191	0.0000	0.0000	0.0806	0.0526	0.0270	0.0667	0.0394	0.1569
s	0.1148	0.0000	0.1000	0.0323	0.0000	0.0000	0.0000	0.0787	0.0000
t	0.0273	0.0000	0.1000	0.0161	0.0000	0.0000	0.0000	0.0236	0.1176
u	0.0027	0.1053	0.0000	0.0081	0.0526	0.1351	0.0267	0.0000	0.0196
w	0.1093	0.0000	0.1000	0.0161	0.0000	0.0270	0.0133	0.0197	0.0000
x	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
y	0.0055	0.0526	0.0000	0.0000	0.0000	0.1622	0.0133	0.0000	0.1176
æ	0.0137	0.0000	0.0000	0.0000	0.0526	0.0000	0.0533	0.0000	0.1765

ð	0.0109	0.0000	0.0000	0.0403	0.0000	0.0270	0.0000	0.0039	0.0000
þ	0.0710	0.0000	0.0000	0.0242	0.0000	0.0000	0.0000	0.0079	0.0000

	g	h	i	l	m	n	o	r	s
	0.0923	0.0303	0.0000	0.2222	0.4805	0.3265	0.0600	0.1400	0.2979
5	0.0000	0.0152	0.0125	0.0000	0.0130	0.0000	0.0100	0.0000	0.0000
7	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
a	0.1385	0.0152	0.0375	0.1270	0.1039	0.0748	0.0000	0.1200	0.0213
b	0.0000	0.0000	0.0125	0.0000	0.0000	0.0000	0.0100	0.0000	0.0000
c	0.0000	0.0000	0.0625	0.0317	0.0000	0.0204	0.0100	0.0100	0.0638
d	0.0154	0.0000	0.0625	0.1111	0.0000	0.0816	0.1400	0.0300	0.0000
e	0.4462	0.2424	0.0375	0.1746	0.0909	0.2109	0.0000	0.2400	0.1489
f	0.0000	0.0000	0.0250	0.0159	0.0000	0.0068	0.0700	0.0300	0.0000
g	0.0000	0.0000	0.0750	0.0000	0.0000	0.0680	0.0300	0.0200	0.0000
h	0.0000	0.0000	0.1125	0.0000	0.0000	0.0000	0.0100	0.0400	0.0000
i	0.0308	0.1667	0.0000	0.0794	0.0779	0.0816	0.0000	0.0800	0.0638
l	0.0000	0.0000	0.0500	0.0476	0.0000	0.0000	0.0800	0.0100	0.0106
m	0.0000	0.0000	0.0750	0.0317	0.0000	0.0068	0.0400	0.0100	0.0000
n	0.0462	0.0000	0.1625	0.0159	0.0000	0.0204	0.2300	0.0400	0.0213
o	0.0769	0.0000	0.0000	0.0635	0.1169	0.0408	0.0000	0.0300	0.0213
r	0.0769	0.0303	0.0000	0.0159	0.0000	0.0068	0.2000	0.0100	0.0000
s	0.0154	0.0000	0.0625	0.0000	0.0000	0.0000	0.0100	0.0100	0.0319
t	0.0000	0.1970	0.0125	0.0000	0.0000	0.0068	0.0500	0.0100	0.1596
u	0.0308	0.0303	0.0000	0.0317	0.0390	0.0136	0.0000	0.0600	0.0106
w	0.0000	0.1515	0.0000	0.0159	0.0000	0.0000	0.0000	0.0100	0.0957
x	0.0000	0.0000	0.0125	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
y	0.0308	0.0455	0.0000	0.0159	0.0130	0.0068	0.0000	0.0100	0.0213

æ	0.0000	0.0303	0.0000	0.0000	0.0649	0.0068	0.0000	0.0500	0.0319
ð	0.0000	0.0455	0.1625	0.0000	0.0000	0.0136	0.0300	0.0300	0.0000
þ	0.0000	0.0000	0.0250	0.0000	0.0000	0.0068	0.0200	0.0100	0.0000

	t	u	w	x	y	æ	ð	þ
	0.3108	0.0500	0.0139	0.0000	0.0000	0.0000	0.5238	0.0222
5	0.0000	0.0250	0.0000	0.0000	0.0000	0.0000	0.0000	0.0222
7	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0222
a	0.0270	0.0000	0.1528	0.0000	0.0000	0.0000	0.1667	0.0889
b	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
c	0.0000	0.0000	0.0000	0.0000	0.0000	0.0328	0.0000	0.0000
d	0.0000	0.0000	0.0000	0.0000	0.0938	0.0820	0.0000	0.0000
e	0.2568	0.0000	0.1667	0.0000	0.0000	0.0000	0.0952	0.2667
f	0.0135	0.0250	0.0000	0.0000	0.0000	0.0328	0.0238	0.0000
g	0.0135	0.0250	0.0000	0.0000	0.0000	0.1148	0.0000	0.0444
h	0.0000	0.0000	0.0000	0.0000	0.0312	0.0328	0.0000	0.0000
i	0.0270	0.0000	0.2778	0.0000	0.0000	0.0000	0.0000	0.0222
l	0.0000	0.0250	0.0139	0.0000	0.1250	0.0328	0.0476	0.0000
m	0.0000	0.4750	0.0000	0.0000	0.0312	0.0656	0.0000	0.0444
n	0.0000	0.1500	0.0000	0.0000	0.1875	0.0164	0.0238	0.0000
o	0.2162	0.0000	0.0694	1.0000	0.0000	0.0000	0.0476	0.1111
r	0.0541	0.0750	0.0278	0.0000	0.3750	0.0984	0.0000	0.0222
s	0.0000	0.0250	0.0000	0.0000	0.0312	0.2295	0.0000	0.0000
t	0.0135	0.0000	0.0000	0.0000	0.0312	0.1803	0.0000	0.0000
u	0.0541	0.0000	0.0139	0.0000	0.0000	0.0000	0.0476	0.0444
w	0.0135	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
x	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

y	0.0000	0.0000	0.0556	0.0000	0.0000	0.0000	0.0000	0.0222
æ	0.0000	0.0000	0.2083	0.0000	0.0000	0.0000	0.0000	0.2444
ð	0.0000	0.0500	0.0000	0.0000	0.0938	0.0164	0.0000	0.0222
þ	0.0000	0.0750	0.0000	0.0000	0.0000	0.0656	0.0238	0.0000

A.4 RNN CHARACTERS TRANSITION MATRIX

	5	7	a	b	c	d	e	f
	0.0024	0.5545	0.0000	0.3286	0.0000	0.1490	0.2499	0.3136	0.0263
5	0.0229	0.0000	0.0000	0.0063	0.0000	0.0438	0.0000	0.0000	0.0127
7	0.0244	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
a	0.0129	0.0000	0.0000	0.0000	0.0751	0.0003	0.1018	0.1201	0.0348
b	0.0384	0.0000	0.0000	0.0066	0.0665	0.0000	0.0000	0.0059	0.0000
c	0.0212	0.0000	0.0000	0.0402	0.0000	0.0000	0.0000	0.0195	0.0000
d	0.0473	0.0000	0.0000	0.0158	0.0000	0.0000	0.0147	0.0260	0.0132
e	0.0141	0.1152	0.0957	0.0000	0.4386	0.2662	0.3600	0.0000	0.2162
f	0.0636	0.0000	0.3632	0.0062	0.0000	0.0001	0.0000	0.0184	0.0000
g	0.0746	0.0561	0.0817	0.0066	0.0000	0.0001	0.0000	0.0205	0.0000
h	0.1246	0.0572	0.0000	0.0233	0.0000	0.0001	0.0000	0.0117	0.0000
i	0.0073	0.0605	0.0000	0.0000	0.1200	0.0004	0.0136	0.0000	0.0000
l	0.0207	0.0000	0.0841	0.0708	0.0000	0.0001	0.0134	0.0623	0.0000
m	0.0751	0.0000	0.0000	0.0312	0.0000	0.0002	0.0000	0.0281	0.0000
n	0.0597	0.0000	0.0681	0.2532	0.0000	0.0921	0.0139	0.1016	0.0000
o	0.0203	0.0000	0.0000	0.0000	0.1070	0.0653	0.0545	0.0820	0.0754
r	0.0185	0.0000	0.0000	0.0814	0.0643	0.0495	0.0687	0.0440	0.1689
s	0.1146	0.0000	0.0816	0.0303	0.0000	0.0001	0.0000	0.0769	0.0000
t	0.0255	0.0000	0.0913	0.0140	0.0000	0.0005	0.0000	0.0283	0.1233
u	0.0018	0.0990	0.0000	0.0062	0.0563	0.1061	0.0256	0.0000	0.0130
w	0.1096	0.0000	0.1342	0.0142	0.0000	0.0505	0.0135	0.0244	0.0000
x	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
y	0.0044	0.0575	0.0000	0.0000	0.0000	0.1297	0.0135	0.0000	0.1201
æ	0.0122	0.0000	0.0000	0.0000	0.0721	0.0004	0.0567	0.0000	0.1961

ð	0.0108	0.0000	0.0000	0.0413	0.0000	0.0450	0.0000	0.0059	0.0000
þ	0.0730	0.0000	0.0000	0.0237	0.0000	0.0002	0.0000	0.0109	0.0000

	g	h	i	l	m	n	o	r	s
	0.0856	0.0334	0.0000	0.2390	0.5045	0.3490	0.0620	0.1442	0.2566
5	0.0000	0.0146	0.0124	0.0000	0.0090	0.0000	0.0092	0.0000	0.0000
7	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
a	0.1518	0.0162	0.0355	0.1502	0.1098	0.0774	0.0000	0.1315	0.0365
b	0.0000	0.0000	0.0123	0.0000	0.0000	0.0000	0.0100	0.0000	0.0000
c	0.0000	0.0000	0.0623	0.0265	0.0000	0.0187	0.0098	0.0085	0.0589
d	0.0126	0.0000	0.0692	0.1365	0.0000	0.0899	0.1536	0.0321	0.0000
e	0.4746	0.2399	0.0341	0.1772	0.0817	0.2027	0.0000	0.2413	0.1435
f	0.0000	0.0000	0.0234	0.0086	0.0000	0.0049	0.0666	0.0282	0.0001
g	0.0000	0.0000	0.0741	0.0000	0.0000	0.0690	0.0310	0.0193	0.0001
h	0.0000	0.0000	0.1119	0.0000	0.0000	0.0000	0.0103	0.0399	0.0000
i	0.0255	0.1727	0.0000	0.0799	0.0767	0.0816	0.0000	0.0826	0.0710
l	0.0000	0.0000	0.0503	0.0419	0.0000	0.0000	0.0755	0.0082	0.0191
m	0.0000	0.0000	0.0756	0.0244	0.0000	0.0051	0.0408	0.0089	0.0000
n	0.0425	0.0000	0.1641	0.0087	0.0000	0.0176	0.2164	0.0398	0.0357
o	0.0712	0.0000	0.0000	0.0587	0.1163	0.0378	0.0000	0.0290	0.0329
r	0.0754	0.0313	0.0000	0.0087	0.0000	0.0045	0.2013	0.0086	0.0000
s	0.0104	0.0000	0.0580	0.0000	0.0000	0.0000	0.0098	0.0082	0.0445
t	0.0000	0.1894	0.0110	0.0000	0.0000	0.0047	0.0510	0.0082	0.1192
u	0.0245	0.0285	0.0000	0.0223	0.0309	0.0103	0.0000	0.0554	0.0178
w	0.0000	0.1479	0.0000	0.0087	0.0000	0.0000	0.0000	0.0084	0.0867
x	0.0000	0.0000	0.0102	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
y	0.0260	0.0455	0.0000	0.0087	0.0093	0.0046	0.0000	0.0089	0.0328

æ 0.0000 0.0317 0.0000 0.0000 0.0618 0.0047 0.0000 0.0499 0.0443

ð 0.0000 0.0491 0.1725 0.0000 0.0000 0.0124 0.0318 0.0305 0.0000

þ 0.0000 0.0000 0.0231 0.0000 0.0000 0.0051 0.0208 0.0084 0.0000

t u w x y æ ð þ

0.3268 0.0471 0.0073 0.0000 0.0000 0.0000 0.5086 0.0185

5 0.0000 0.0236 0.0000 0.0000 0.0000 0.0000 0.0000 0.0191

7 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0198

a 0.0231 0.0000 0.1592 0.0000 0.0000 0.0000 0.1725 0.0905

b 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000

c 0.0000 0.0000 0.0000 0.0000 0.0000 0.0394 0.0000 0.0000

d 0.0000 0.0000 0.0000 0.0000 0.0918 0.0916 0.0000 0.0000

e 0.2604 0.0000 0.1532 0.0000 0.0000 0.0000 0.1018 0.2731

f 0.0089 0.0268 0.0000 0.0000 0.0000 0.0374 0.0230 0.0000

g 0.0096 0.0296 0.0000 0.0000 0.0000 0.1111 0.0000 0.0457

h 0.0000 0.0000 0.0000 0.0000 0.0107 0.0416 0.0000 0.0000

i 0.0228 0.0000 0.3123 0.0000 0.0000 0.0000 0.0000 0.0164

l 0.0000 0.0249 0.0093 0.0000 0.1241 0.0389 0.0479 0.0000

m 0.0000 0.4574 0.0000 0.0000 0.0110 0.0700 0.0000 0.0448

n 0.0000 0.1544 0.0000 0.0000 0.1842 0.0239 0.0242 0.0000

o 0.2340 0.0000 0.0600 1.0000 0.0000 0.0000 0.0493 0.1121

r 0.0490 0.0739 0.0259 0.0000 0.5032 0.1001 0.0000 0.0165

s 0.0000 0.0289 0.0000 0.0000 0.0096 0.1928 0.0000 0.0000

t 0.0095 0.0000 0.0000 0.0000 0.0108 0.1606 0.0000 0.0000

u 0.0468 0.0000 0.0075 0.0000 0.0000 0.0000 0.0469 0.0362

w 0.0091 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000

x 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000

y	0.0000	0.0000	0.0515	0.0000	0.0000	0.0000	0.0000	0.0170
æ	0.0000	0.0000	0.2137	0.0000	0.0000	0.0000	0.0000	0.2687
ð	0.0000	0.0495	0.0000	0.0000	0.0546	0.0231	0.0000	0.0216
þ	0.0000	0.0840	0.0000	0.0000	0.0000	0.0696	0.0258	0.0000

A.5 The Python code

A.5.1 Listing 1: Input length 1, Output length 1

```
#!/usr/bin/env python
# coding: utf-8

# In[23]:



import numpy as np
import string
import random
import pandas as pd

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Activation
from tensorflow.keras.layers import Input, Embedding
from tensorflow.keras.layers import SimpleRNN
from tensorflow.keras.callbacks import ModelCheckpoint
from keras.utils import np_utils
from tensorflow.python.keras import backend as k

import matplotlib.pyplot as plt
```

```
# In[24]:


'''
prepare_vocab method:
1. reads and cleans the old english data: removes punctuations, converts
characters and makes a set of vocabulary.
2. One-hot-vector encode: converts the characters to number for inputtinc
3. Converts the numbers back into characters to output the predicted cha
'''
def prepare_vocab(filename, prediction_num=1):
    #Part one: read and clean data
    with open(filename, 'r', encoding='ANSI') as file:
        data = file.read()
        data = data.lower()
        data = data.translate(str.maketrans('', '', string.punctuation))
        data = data.replace('\n', ' ')

    #Part 2: Make vocabulary list
    chars = sorted(list(set(data)))
    print('Vocabulary: ', chars)
    print ('Vocabulary Size: ', len(chars))

    #Part 3: One hot vector encode: convert characters to numbers
    char_to_index = {v:i for i,v in enumerate(chars)}
    index_to_char = {i:v for i,v in enumerate(chars)}
```

```
    total_index = [char_to_index[char] for char in data]

    # total_index
    ''.join(index_to_char[i] for i in total_index)

    #Part 4: Convert numbers to characters, identify inputs and outputs
    pred_num = prediction_num

    '''
    create an array of the first n characters, where 0 to (n-1) characte
    xin = 0 to (n-1) characters
    yin = nth character
    '''
    xin = [[total_index[j+i] for j in range(0, len(total_index)-1-pred_nu

    yin = [total_index[i+pred_num] for i in range(0, len(total_index)-1-p
    return chars, xin, yin, char_to_index


# In[25]:


def compute_atransitions(filename, vocab):
    file = open(filename, 'r', encoding='ANSI')
    data = file.read()
    data = data.lower()
```

```
    data = data.translate(str.maketrans('', '', string.punctuation))
    data = data.replace('\n', ' ')
    my_preds = {}
    probs = [0] * len(vocab)
    for c in vocab:
        my_preds[c] = {nxt_chr: prob for (nxt_chr, prob) in zip(vocab, pr

    for i in range(1,len(data)):
        pc = data[i-1]
        cc = data[i]
        my_preds[pc][cc] += 1
    counts_df = pd.DataFrame(my_preds)
    return counts_df


# In[26]:


'''
build_model function:
1.To build the model I use the Embedding, SimpleRNN and Dense layers fro
2. The input/first layer is the Embedding layer, where the input length i
that will be predicted
3. The hidden/second layer is SimpleRNN layer with relu as the activatio
4. The output/final layer is dense layer and it outputs prediction of the
is softmax.
```

```
5. The optimizer for this model is adam
'''
def build_model(vocabulary, prediction_num, hidden_layers=128, n_fac=8):
    vocab_size = len(vocabulary)

    #Creating the model
    model = Sequential([
            Embedding(vocab_size, n_fac, input_length=prediction_num), #I
            SimpleRNN(hidden_layers, activation='relu'), #Hidden layers
            Dense(vocab_size, activation='softmax') #output layers
        ])

    model.compile(loss='sparse_categorical_crossentropy', optimizer='adam
    model.summary()

    return model

'''
Function train_model:
1. Uses the model from the build model function and trains it on the expe
prepare vocab function
2. Fits the model on the given input and out
3. For arrays X and Y, we are removing the last 2 characters to keep the
'''
def train_model(x, y, model, pred_num, batch_size, epochs):
    X = [np.stack(x[i][:-2]) for i in range(pred_num)]
```

```
    Y = np.stack(y[:-2])

    model.fit(np.stack(X, 1), Y, batch_size=batch_size, epochs=epochs)
    return model


# In[27]:


'''
predict_next_car function:
1.First convert the input, inp to indices
2. Then expand the dimension to match the model's output format
3. Predict the nth character using the input
4. As we are using softmax activation in the last layer of the model, we
our vocabulary. So the character with the maximum probability will be the
'''
def predict_next_char(inp, model_, c2i_map):
    index = [c2i_map[i] for i in inp]
    arr = np.expand_dims(np.array(index), axis=0)

    prediction = model_.predict(arr)
    return prediction


'''
```

```
generate probes function
1. Extracts the probabilities associated with each predicted characters
2. Maps the probabilities to the corresponding characters
3. Saves them in a dictionary
'''
def generate_probs(vocab, model_, c2i_map):
    my_preds = {}
    for c in vocab:
        preds = predict_next_char(c, model_, c2i_map)
        my_preds[c] = {nxt_chr: prob for (nxt_chr, prob) in zip(vocab, p
    return my_preds


# In[28]:


np.random.seed(3791)

EPOCHS = 85
BATCH_SIZE = 8
PRED_NUM = 1 #number of characters you want to predict

vocab, x, y, char_to_index = prepare_vocab(r'C:\Users\Sushmita\Desktop\T
model = build_model(vocab, PRED_NUM) #call build_model function with para

model = train_model(x, y, model, PRED_NUM, BATCH_SIZE, EPOCHS) #call the
```

```
model.save_weights('simpleRNN_3pred.h5') #save the weights


# In[22]:


#Call the function generate probability passing the model, vocab and one-
next_char_probs = generate_probs(vocab, model, char_to_index)
prob_df = pd.DataFrame(next_char_probs) #Save the output in a df
display(prob_df) #Display the dataframe


# In[8]:


prob_df[vocab[0:9]].to_csv('transition1.csv',sep='\t',
                index=True, float_format='%.4f', encoding='ANSI') #saving
prob_df[vocab[9:18]].to_csv('transition2.csv',sep='\t',
                index=True, float_format='%.4f', encoding='ANSI') #saving
prob_df[vocab[18:]].to_csv('transition3.csv',sep='\t',
                index=True, float_format='%.4f', encoding='ANSI') #saving


# In[9]:
```

```
#collect the counts and apriori probabilities
counts_df = compute_atransitions(r'C:\Users\Sushmita\Desktop\Thesis\old_e
aprob_df = counts_df.copy()
char_counts = {}
for j in range(counts_df.shape[1]):
    s = sum(aprob_df.iloc[:,j])
    char_counts[vocab[j]] = s
    for i in range(counts_df.shape[0]):
        if (s > 0):
            aprob_df.iloc[i,j] = aprob_df.iloc[i,j] / s


# In[10]:


aprob_df[vocab[0:9]].to_csv('atransition1.csv',sep='\t',
               index=True, float_format='%.4f', encoding='ANSI')
aprob_df[vocab[9:18]].to_csv('atransition2.csv',sep='\t',
               index=True, float_format='%.4f', encoding='ANSI')
aprob_df[vocab[18:]].to_csv('atransition3.csv',sep='\t',
               index=True, float_format='%.4f', encoding='ANSI')


# In[11]:
```

```
#%%------------ next char table (decreasing order of probabilities)
next_sequenceRNN = {} #create dictionary of the sequence generated by RNN
next_sequenceHMM = {} #create dictionary of the sequence generated by RNN
next_sequence = {}

err_cnt = 0
for c in vocab:
    a1 = prob_df.sort_values(by=c,ascending=False)[c].index.values #sort
    a2 = aprob_df.sort_values(by=c,ascending=False)[c].index.values
    err_cnt += sum(a1 != a2)
    next_sequenceRNN[c] = ''.join(a1)
    next_sequenceHMM[c] = ''.join(a2)
    next_sequence['[RNN] '+c] = ''.join(a1)
    next_sequence['[HMM] '+c] = ''.join(a2)


# In[12]:


err_cnt


# In[13]:


#%%------------ some tests --------------------------
```

```
top = np.array(prob_df['a']).argsort()[-5:][::-1]
np.array(vocab)[top]
#%%========================== RESULTS ==================================
#characters frequencies
plt.style.use("ggplot")
plt.figure()
plt.bar(vocab, char_counts.values(), align='center', alpha=0.5)
plt.title("Characters frequencies")
plt.xlabel("character")
plt.ylabel("counts")

#%%--------------- some plots of output/transition probabilities---------
char = '5'
plt.style.use("ggplot")
plt.figure()
plt.plot(prob_df[char], label="RNN transitions")
plt.plot(aprob_df[char], label="Apriori transitions")
plt.title("RNN vs. Apriori transitions for character ['" + char + "']")
plt.xlabel("follows ['" + char + "']")
plt.ylabel("probability")
plt.legend(loc="upper right")
```

A.5.2 LISTING 2: INPUT LENGTH K ($K \geq 2$), OUTPUT LENGTH 1

```
#!/usr/bin/env python
# coding: utf-8
```

```
#%%----------------------- libraries ------------------------------


import numpy as np
import string

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import SimpleRNN

import matplotlib.pyplot as plt

# In[37]:


def prepare_vocab(filename, prediction_num=1, backwards = False):
    with open(filename, 'r', encoding='ANSI') as file:
        data = file.read()
        data = data.lower()
        data = data.translate(str.maketrans('', '', string.punctuation))
        data = data.replace('\n', ' ')

        if (backwards):
            data = ''.join(reversed(data))
```

```
    chars = sorted(list(set(data)))
    print('Vocabulary: ', chars)
    print ('Vocabulary Size: ', len(chars))

    # chars.insert(0, '\0')
    char_to_index = {v:i for i,v in enumerate(chars)}
    index_to_char = {i:v for i,v in enumerate(chars)}
    total_index = [char_to_index[char] for char in data]

    # total_index
    ''.join(index_to_char[i] for i in total_index)

    # char_to_index
    pred_num = prediction_num
    data1 = np.array(total_index)
    #xin = [[total_index[j+i] for j in range(0, len(total_index)-1-pred_
    #yin = [total_index[i+pred_num] for i in range(0, len(total_index)-1
    data1 = np.append(data1,np.repeat(data1[-1,],pred_num))
    xin, yin = convertToMatrix(data1, pred_num)
    xin = np.reshape(xin, (xin.shape[0], 1, xin.shape[1]))

    return chars, xin, yin, char_to_index, index_to_char, data1


def convertToMatrix(data, prediction_num):
```

```
    X, Y =[], []
    for i in range(len(data)-prediction_num):
        d=i+prediction_num
        X.append(data[i:d,])
        Y.append(data[d,])
    return np.array(X), np.array(Y)


# In[38]:


def build_model(vocabulary, prediction_num, hidden_layers=256, n_fac=42)
    vocab_size = len(vocabulary)

    model = Sequential([
            SimpleRNN(units=hidden_layers, input_shape=(1,prediction_num)
            Dense(vocab_size, activation='softmax')
        ])

    model.compile(loss='sparse_categorical_crossentropy', optimizer='adam
    model.summary()

    return model

# In[39]:
```

```
def predict_next_char(inp, model_, c2i_map, i2c_map, pred_num):
    inp = inp + ' '
    index = np.array([c2i_map[i] for i in inp])
    arr, y = convertToMatrix(index, pred_num)
    arr = np.reshape(arr, (arr.shape[0], 1, arr.shape[1]))

    pred = model_.predict(arr)
    idx = np.argmax(pred)
    return i2c_map[idx]


def predict_next(inp, model_, c2i_map, i2c_map, pred_num):
    inp = inp + ' '
    index = np.array([c2i_map[i] for i in inp])
    arr, y = convertToMatrix(index, pred_num)
    arr = np.reshape(arr, (arr.shape[0], 1, arr.shape[1]))

    pred = model_.predict(arr)[0]
    s = np.array(pred).argsort()[::-1]
    seq = {}
    for i in s:
        c = i2c_map[i]
        seq[c] = pred[i]

    return seq
```

```
def generate_counts(inp, vocab, data, c2i_map, i2c_map):
    my_preds = {}
    for c in vocab:
        my_preds[c] = 0

    arr = np.array([c2i_map[i] for i in inp])
    la = len(arr)
    #lazy search
    for i in range(len(data)-la):
        #check if arr found at this position
        found = True
        for j in range(la):
            if (arr[j] != data[i+j]):
                found = False
                break
        if (found):
            c = i2c_map[data[i+la]]
            my_preds[c] += 1



    return my_preds

def generate_probs(inp, vocab, data, c2i_map, i2c_map):
    counts = generate_counts(inp, vocab, data, char_to_index, index_to_cl
    s = sum(counts.values())
```

```
    probs = counts.copy()
    if (s > 0):
        for k,v in probs.items():
            probs[k] = v/s

    return probs

def HMM_next(inp, vocab, data, c2i_map, i2c_map):
    pred = generate_probs(inp, vocab, data, char_to_index, index_to_char
    return sorted(pred.items(), key=lambda kv: kv[1], reverse=True)

# In[40]:
np.random.seed(2020)

EPOCHS = 150
BATCH_SIZE = 8
PRED_NUM = 3
PRED_SEQ_LEN = 1
BOOSTFACTOR = 5

vocab, x, y, char_to_index, index_to_char, data = prepare_vocab(r'data\o
if (BOOSTFACTOR > 1):
    x = np.concatenate([x for i in range(BOOSTFACTOR)], 0)
    y = np.concatenate([y for i in range(BOOSTFACTOR)], 0)
```

```
model = build_model(vocab, PRED_NUM)

model.fit(x,y, epochs=EPOCHS, batch_size=BATCH_SIZE, verbose=2)
#model.save_weights(r'data\simpleRNN_3pred.h5')

inp = 'nge'
predict_next_char(inp, model, char_to_index, index_to_char, PRED_NUM)
predict_next(inp, model, char_to_index, index_to_char, PRED_NUM)
HMM_next(inp, vocab, data, char_to_index, index_to_char)
```