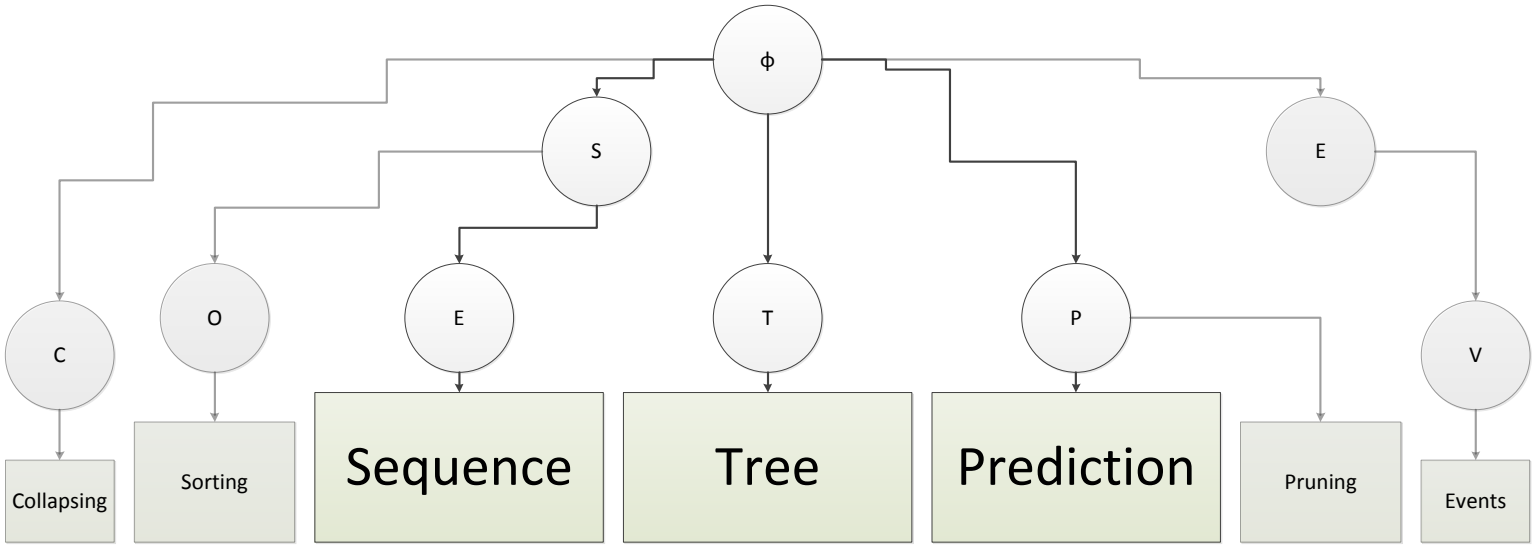

Frederik Frandsen (ffrand09@student.aau.dk)
Mikkel Hansen (mfha09@student.aau.dk)
Peder Sørensen (psoere09@student.aau.dk)
Aalborg University, Denmark
June 9, 2011

Aalborg University
Department of Computer Science

TITLE:
Sequence Tree Prediction

PROJECT PERIOD:
February 1^{st}, 2011
June 9^{th}, 2011

PROJECT GROUP:
f11d620a

GROUP MEMBERS:
Frederik Frandsen
Mikkel Hansen
Peder Sørensen

SUPERVISORS:
Yifeng Zeng

NUMBER OF COPIES: 5

TOTAL PAGES: 93

SYNOPSIS:

In this report we look at sequence prediction in the context of predictive text. We propose a method called Sequence Trees and define it formally. We also implement and test several optimizations to improve the space and time complexity of our method. Sequence trees are finally evaluated using two other methods n-grams and classification trees as benchmark methods on the domain of predictive text.

The sequence tree predictor's accuracy within the predictive text domain is better than its classification tree and n-gram counterparts. While its space requirement is a limiting factor the method requires significantly less space than classification trees. The n-gram predictor uses even less space, but also has a lower prediction accuracy than sequence trees. Using the suggested optimizations we however achieve both high prediction and low spacial requirements in the domain of predictive text.

CONTENTS

CHAPTER

ONE

INTRODUCTION

Sequence learning is the task of learning through the observation of sequences. Many of the tasks we ourselves perform rely on some form of sequence learning; Even simple tasks can be seen as sequence learning. Walking for examples is a sequence of lifting each leg in the correct order to maintain balance and move forward. Sun et al.[17] state that humans often acquire skills by way of studying sequential actions, this can include the above example of walking or complex tasks such as playing an instrument. Anderson[1] argues that problem solving and reasoning often requires sequential learning as well. Intuitively, some machine intelligence tasks could also be more efficiently learned or directly require some form of sequence learning.

Such machine intelligence tasks include inference, planning, reasoning, natural language processing, speech recognition, financial engineering, DNA sequencing and other examples given by Sun & Giles[19]. All of these different tasks give rise to four related categories that are all aspects of sequence learning:

- **Sequence prediction**
  Here previously seen elements of a sequence are used to predict the next elements of that sequence.

- **Sequence generation**
  Here previous sequence elements are used to generate the next elements of that sequence.

- **Sequence recognition**
  Here elements of a sequence are examined to decide if the sequence complies with some criteria.

- **Sequential decision making**
  Here a sequence of actions are selected to complete a goal.

For this project we focus on sequence prediction and recognition. We attempt to recognize sequences by looking at their elements and the order of these elements in each sequence, and when a sequence is recognized we can return an identifying label of that sequence. Each such label is an abstraction over an entire sequence of elements, and will be described further later.

We propose the use of sequence trees for storing ordered sequence elements and identifying labels of that sequence. The details of the sequence tree prediction method and where it differs from other prediction solutions will be detailed throughout this report.

First a general definition of sequence trees will be given, along with prediction accuracy improvements and methods for improving space and time requirements of the method. We will then test sequence trees in the domain of mobile device predictive text to determine whether sequence trees are viable as a sequence prediction method.

1.1 Related Work

There are many approaches to sequence prediction, and most of these are classical methods such as using neural networks, hidden Markov models and reinforcement learning as explained by Sun & Giles[19].

Frandsen et al.[5] applied action trees, a method that sequence trees are inspired by, to prediction of an opponent's moves in computer games, and using sequences of player actions revealed itself to be a valid approach to prediction. Initial tests showed action trees to perform better than multilayer perceptrons and naive Bayes classifiers within this opponent modeling domain, but action trees' performance outside this domain has not yet been examined. Furthermore, the test data used proved insufficient in size for covering the complete state space of the video game opponent modeling domain.

Other methods similar to sequence trees is classification trees described by Kohavi et al. in [10] and Jensen et al.[9], also known as decision trees in some literature such as Tom Mitchell[14]. This method is already used in many fields such as botany, medicine, computer science and psychology. An example of the use of classification trees is SPATTER by Magerman[13], a tool for natural language processing that analyzes English written sentences to derive phrases, verbs, nouns and their overall structure. Another example of natural language processing by Ezra Black et al.[4] is grammatical tagging, where they use classification trees to predict whether some word in a written sentence is a noun, verb or other category, based on adjacent words in the sentence.

Another method that shares some of the properties of sequence trees is n-grams. N-grams have previously been used for predictive text, by Bickel et al.[3] where n-grams are used to predict sentences. Another approach to predictive text using n-grams has been done by Jon Hasselgren et al.[6], where bi-grams were used to disambiguate the predictions of the next word based on the previous. Additionally, n-grams have been compared to classification trees by Häkkinen and Tian[7] for use in identifying the language of short sentences and names. More generally n-grams has applications within natural language prediction and speech recognition, handwriting recognition etc.

Several predictive text methods have already been developed and are in widespread use today, such as the T9 dictionary disambiguation for writing text on mobile phones examined by MacKenzie [11]. Another approach also examined by Mackenzie is LetterWise [12]. LetterWise is a unique approach for predictive text

using prefix based disambiguation, i.e. using the probability of seeing a letter based on the previous letter. Many of the common predictive text approaches are however closed source and it is not possible to gain information on their inner workings.

A few open source predictive text solutions exists such as Presage[2] are also publicly available, but not all of these are in a widespread use. There has been much work done on predictive text for mobile phones as well, but most of this research is done internally by phone companies and is not available in public. There are however some benchmarks available on how these predictors perform, such as those given by MacKenzie[11].

1.2 Predictive Text

Today many computer programs with user input make use of prediction to improve operation efficiency and user experience. Several web browsers such as Google Chrome, Internet Explorer, Mozilla Firefox and more all use prediction, or auto completion, when users input website addresses.

Similarly, many online or offline search engines such as Google Search, Windows Search and more use prediction to provide search results to the user more quickly. Programmer's development environments such as Visual Studio and Eclipse also make use of autocompletion to assist users in designing programs faster.

These examples and many more fall within the domain of predictive text; that is, generation of text from some input by way of prediction. One example of such could be a user intending to write "*house*" in an online search engine. So far only the letters h, o and u have been given as input, but the search engine may already be able to provide a range of suggested words to the user, such as "*hour*", "*hound*" and "*house*".

The quality of the suggested words depends on the text prediction method used; in this case "*house*" was only suggested as the third word rather than the first. Another better text prediction method might have been able to suggest "*house*" as the first word rather than the third. The user now has to cycle through the two other suggested words to reach "*house*" and complete his or her search query.

The best text prediction method is then the one which allows the user to input information in the system in the shortest amount of time or effort. A measurement of such writing efficiency is keystrokes per character, as presented by MacKenzie[11]:

Definition 1. *Key-Strokes Per Character (KSPC)*
KSPC is the number of keystrokes required, on average, to generate a character of text for a given text entry technique in a given language.

The basic way to calculate KSPC, given by MacKenzie [11], for writing a word is:

$$KSPC = \frac{\sum_w (K_w * F_w)}{\sum_w (C_w * F_w)} \tag{1.1}$$

Where K_w is the keystrokes needed to write word w, C_w is the number of characters in w, and finally F_w is the frequency of w in a word corpus. We will use two permutations of this equation in Section 7.7.1 to represent keypad and touchscreen input methods.

For regular "qwerty" keyboards the KSPC is 1 for writing sentences of lower case letters, as there is a button for each letter. However, mobile phones and other devices might designate one button to multiple letters in the alphabet to reduce keypad size, which results in ambiguous text input. A high KSPC is a sign of an ineffective input method. A KSPC below one means that the input method predicts some of the characters in the words written.

1.3 Ambiguous Input

On most mobile devices there is only a limited number of space for keyboard buttons. Most phones usually only have nine buttons to use for writing text. Each letter is encoded to a number on the device, depending on the language. An illustration of this can be seen in Figure 1.1.

Figure 1.1: An illustration of a mobile phone. The keyboard is only 12 buttons, with the alphabet distributed over the buttons as shown.

This "numberfied" mapping of 26 letters onto a mobile phone's 9 number buttons produces ambiguous text input; now one button may correspond to three or four different letters. One of the most simple solutions to selecting between several letters per button is multi-tap disambiguation where a button has to be pressed multiple times in order to get a specific letter. An example could be a user attempting to write "*cow*" in a mobile phone text message. The user would, when using the multi-tap input mode, have to input the number sequence 2226669 to write "*cow*", which is significantly more troublesome than simply writing "*cow*" on a "qwerty" keyboard. This multi-tap method thus reduces the speed at which one can write text considerably.

Furthermore, consecutive identical letters from the same encoding group need to be separated using another keystroke or by waiting for a timeout to pass. For instance, writing the word "*oops*" using multi-tap would require the keystroke sequence 666_6667_7777, where each underscore is the user waiting for a timeout or pressing an additional special key.

This rudimentary multi-tap text input system can have a high KSPC (often above 2) for most input tasks. If we return to the word "oops", we can find the

KSPC using Equation 1.1. If we count the number of keystrokes in the "oops" example above, we need 13 keystrokes when we count the underscore as an additional keystroke. We divide the amount of keystrokes with the size of the word, which is four. The calculation can be seen in equation 1.2, where we have set the frequency to one.

$$\frac{K_{oops} \times F_{oops}}{C_{oops} \times F_{oops}} = \frac{13}{4} = 3.25 \tag{1.2}$$

As the calculation show, and as was intuitively true multi-tapping is not very efficient as we need to press more than three keys for each character. One solution to the low efficiency of multi-tapping is dictionary disambiguation; when a number key is pressed the system suggests a range of words from a dictionary that it predicts is what the user is typing.

Returning to the previous "*cow*" example, after inputting the numbers 2, 6 and 9 several words might be possible, such as "*boy*", "*any*", "*coy*" and "*cow*". A prediction method might here suggest "*cow*" as the first disambiguation of the number sequence 269, while another may choose "*coy*". This disambiguation combined with automatic word completion can in many cases bring the user's KSPC near or even below 1.

Most if not all modern mobile devices use some method to disambiguate input when the user writes text, as it greatly improves the speed that one can write messages over basic multi-tap input. One such established method is called T9 and is found on a very large number of mobile phones. MacKenzie [11] has found the predictive text method T9 to have a KSPC of 1.0072.

As stated above predictive text is mostly researched closed source, and only a few open source projects exists. Thus there is little public knowledge on how to create predictive text methods, and what approaches are effective. We believe that sequence prediction is a very good approach to predictive text as text is highly sequential in nature. It would be interesting to see whether a sequential prediction method would be able to predict written text with high accuracy.

1.4 Project Goals

In this report we develop and examine sequence trees, an approach to general sequence prediction. We will define sequence trees within a general domain and later perform experiments using this method in the domain of predictive text; this test domain offers easily interpretable test data which will allow for rapid testing.

With regards to testing we wish to see whether sequence trees can offer significant improvement to text prediction compared to two existing methods used in sequence prediction: classification trees and n-grams. When comparing these three methods we will test the following aspects:

- Prediction accuracy
- Space requirements of the predictor.
- Average number of Key-Strokes Per Character needed to complete a range of words on a simulated mobile phone keypad.

Prior to this method comparison we will explore the initial performance of sequence trees, and possible means of optimizing this prediction method, as it is still largely untested and may have room for further performance improvements. The main areas of optimization to be explored and tested are:

- Sequence tree storage space reduction.
- Sequence tree prediction speed increase.

We propose several optimizations for each of the two areas in the report and test to see whether the optimizations are effective at improving our sequence tree prediction method.

1.5 Report Overview

We now provide a general overview of this project and the chapters included in this report:

- **Chapter 2 (Sequence Prediction Theory)**
  Here we examine the general area of sequence prediction, and define existing and new concepts needed for describing the sequence prediction methods used in this report.
- **Chapter 3 (Related Techniques for Prediction)**
  Next we go into detail describing classification trees and n-grams, the two prediction methods we will compare with the sequence tree prediction method. We also describe how the two methods are used specifically as text predictors.
- **Chapter 4 (Sequence Trees Construction)**
  This chapter describes sequence trees, the sequence tree construction algorithm, together with four optimizations, tree collapsing, tree pruning and finally clustering to reduce the size of the constructed predictors.
- **Chapter 5 (Sequence Trees Prediction)**
  The next chapter explains how prediction is done using sequence trees. In the chapter we propose two optimizations to the prediction method, tree sorting, and distribution sampling to increase prediction speed and prediction distribution respectively.
- **Chapter 6 (Modeling N-grams with Sequence Trees)**
  Here we look at similarities between sequence trees and n-grams, along with how n-grams can be modeled with the new sequence trees method. We also give a formal proof showing that sequence trees can any model n-gram predictor.
- **Chapter 7 (Sequence Predictor Tests)**
  Next we describe our method of testing and the range of tests performed on sequence trees, along with their results and discuss their significance along with the observed performance of sequence trees.

- **Chapter 8 (Conclusion)**
  Finally we provide a summary of the project and our findings on sequence trees and their use as sequence predictors, along with suggestions for further improving the sequence trees prediction method.

CHAPTER

TWO

SEQUENCE PREDICTION THEORY

In this chapter we will first describe sequence prediction and recognition in general. These definitions will then be used to describe labeling, event sequence prediction and event sequence distribution. These will be used in later chapters when describing sequence trees along with n-grams and classification trees.

2.1 Events and Systems of Events

To discuss our sequence prediction method we will need a number of definitions how we can perform a prediction. In this section we define events which are the elements we predict on, and a system of events that produces the sequences we wish to predict. Lastly we define sequences of events produced by a system.

Definition 2. *An Event*
An event is a possible action that can be observed within a system. In addition to events that can be observed we use the special event ϕ making the initial state of the system prior to any events happening.

Such an event can be anything that makes sense in the current domain. Within the domain of chess strategy prediction one event could be "Rook D2 to D4". In the domain of predictive text events could correspond to letters in the case of word prediction and full words when predicting sentences. An event is not very interesting to predict on if it is not the part of a larger system, however. We define a system of events as in Definition 3.

Definition 3. *Event System*
An event system is a system that produces a number of observable events. We write the set of possible events in a system as $E = \{\phi, e_1, ..., e_n\}$ for any finite n.

Within chess strategy prediction the event system describes all possible chess moves as events, including for instance the "Rook D2 to D4" event. We will only discuss systems with finite sets of events in the following, and now move on to event sequences produced by event systems.

Definition 4. *Event Sequence*
$S = \{s_1, ..., s_j\}$ is an ordered sequence of events where $s_1, ..., s_j \in E$.

An example event sequence for chess strategy prediction is {"Pawn E2 to E3", "Rook D2 to D4"}, and an example event sequence for predictive text is {"h", "o", "u", "s", "e"}. For the sake of brevity we will in the following often write such a letter sequence as simply "house". With events and event sequences now defined we move on to general sequence prediction.

2.2 Sequence Prediction

Sequence prediction is the concept of predicting the most likely next events in a sequence given the previous events in that sequence. Our definition of general sequence prediction is inspired by that of Sun & Giles[18]:

Definition 5. *Sequence Prediction*
General prediction on a sequence $S = \{s_i, ..., s_j\}$ *where* $1 \leq i \leq j \leq \infty$ *is to predict the next* $k \geq 1$ *events* $s_{j+1}, ..., s_{j+k}$ *for that sequence.*

$$\{s_i, ..., s_j\} \rightarrow \{s_{j+1}, ..., s_{j+k}\}$$

In this definition a prediction with $i = 1$ will be based on all the prior events of sequence S, and a prediction with $i = j$ is based on only the prior event s_j. Generally a prediction would only provide the next event s_{j+1} of sequence S, but the following events $s_{j+2}, ..., s_{j+k}$ from the previous definition can also be predicted if we base our next prediction on a previous prediction; this is called open-loop prediction.

An example within predictive text of this open-loop prediction is predicting the letters "se" from the letter sequence "house", where $i = 2$, $j = 3$ and $k = 2$. In this case the letters "ou" from "house" will be used to predict the next letter "s". We then perform another prediction on the letters "us" producing the letter "e". Thus after two prediction iterations we have the predicted letter sequence "se".

We can also perform probabilistic sequence prediction instead of deterministic sequence prediction. In our sequence prediction definition we only return one sequence as a prediction, but we can instead calculate a distribution $p(\{s_{j+1}, ..., s_{j+k}\}|\{s_i, ..., s_j\})$ describing the probability of $\{s_{j+1}, ..., s_{j+k}\}$ being a prediction for $\{s_i, ..., s_j\}$. Where the probability can be calculated according to the general chain rule:

$$\begin{aligned} p(\{s_{j+1}, ..., s_{j+k}\}|\{s_i, ..., s_j\}) \quad = \quad & p(\{s_{j+k}\}|\{s_i, ..., s_j, s_{j+1}, ..., s_{j+k-1}\}) \cdot \\ & ... \cdot \\ & p(\{s_{j+2}\}|\{s_i, ..., s_j, s_{j+1}\}) \cdot \\ & p(\{s_{j+1}\}|\{s_i, ..., s_j\}) \end{aligned} \tag{2.1}$$

2.3 Sequence Recognition

Another aspect of sequence learning we will use is that of sequence recognition. In short, this is determining if a sequence is legitimate based on some criteria. Our definition of general sequence recognition is also inspired by that of Sun & Giles[18]:

Definition 6. *Sequence Recognition*
General recognition of a sequence $S = \{s_i, ..., s_j\}$ *is determining if* S *is legitimate or not, where* $1 \leq i \leq j \leq \infty$.

$$\{s_i, ..., s_j\} \rightarrow \textit{yes or no}$$

At first glance sequence recognition has little to do with prediction, however we can in fact turn it into sequence prediction by basing our recognition on predictions. In other words, we can say that $\{s_i, ..., s_j, s_{j+1}\} \rightarrow yes$ if and only if the predicted s_{j+1} is the correct prediction. Thus we answer *yes* for legitimacy for sequences with predicted events $s_{j+1}, s_{j+2}, ...$ only if all these predicted events match the actual events $s_{j+1}, s_{j+2}, ...$ for the given sequence.

Sequence recognition is here again defined as deterministic, but similar to sequence prediction we can perform probabilistic sequence recognition as well. In this case we calculate the distribution $p(\{s_i, ..., s_j\})$ describing the probability of $\{s_i, ..., s_j\}$ being a legitimate sequence.

2.4 Sequence Labelling

The last thing we need to define before we introduce event sequence prediction is the concept of sequence labels and labeled event sequences. A label can be viewed as the solution to training data in supervised learning schemes, and a labeled event sequence is then an event sequence with an assigned label. The task will then be to assign an event sequence with a fitting label $l \in L$ where L is the set of all possible labels in the domain.

These labels and solutions are domain specific; in our domain of predictive text a label for the letter sequence "house" observed during training could simply be the word "house" itself, but in other domains a more abstract label may be necessary. In chess strategy prediction, for instance, it may be more informative to label a sequence of moves observed during training as the "Sicilian Defense" opening strategy rather than listing each move in sequence. The exact definition of a labeled event sequence is given in the following:

Definition 7. *Labeled Event Sequence*
A labeled event sequence $S_l = (\{s_i, ..., s_j\}, l)$ is an event sequence with an associated label $l \in L$, where $1 \leq i \leq j \leq \infty$.

Consider as an example a labeled event sequence for chess strategy prediction ({"Pawn C7 to C5", "Pawn D7 to D6", ...}, "Sicilian Defense"), with the label being "Sicilian Defense". Only the first two moves are shown in this example for the sake of brevity. Other sequences of chess moves corresponding to a "Sicilian Defense" strategy also exist, and a given chess strategy predictor should be able to label these sequences as a "Sicilian Defense" as well.

2.5 Predicting Sequences of Events

With all of the prior definitions in place we can now provide our definition of event sequence prediction. It is based on the general sequence recognition described earlier, with some events s_{j+1} to s_k in the recognized sequence being predictions instead of actual events.

Definition 8. *Event Sequence Prediction*
Event sequence prediction is a mapping from an event sequence $S = \{s_i, ..., s_j, s_{j+1}, ..., s_{j+k}\}$ to a label $l \in L$ where $1 \leq i \leq j \leq \infty$ and $k \geq 1$. $\{s_i, ..., s_j, s_{j+1}, ..., s_{j+k}\} \rightarrow l$

Returning to the "house" predictive text example, two labels "house" and "shoe" could have been observed during training; in this case the label "house" would resemble the predicted sequence "house" more than the "shoe" label and therefore be selected as the label for that sequence. In the domain of predictive text one way to calculate the resemblance of two labels is given as:

Definition 9. *Resemblance measurement of text*
For text strings we measure resemblance as the number of identical characters in each position in the strings.

Given tree strings "house", "mouse", and "more" we can calculate which are more similar. Both "house" and "mouse" has identical characters at four positions, "more" only has two characters at the same positions with the other words. We can then say that "house" resembles "mouse" more than it does the word "more".

For a given deterministic event sequence prediction on sequence S the predictor only returns one label $l \in L$. Which label $l \in L$ the predictor should return depends on which label most closely resembles sequence S. This is largely up to the implementation, but can be thought of as creating a recognizer for each label $l \in L$ answering *yes* or *no* according to the following:

Definition 10. *Event Sequence Label Recognition*
Event sequence $S = \{s_i, ..., s_j, s_{j+1}, ..., s_{j+k}\}$ with label l is legitimate if and only if l resembles S more than any other label in L.

$$(S, l) \rightarrow \begin{cases} yes & \text{if } l \text{ resembles } \{s_i, ..., s_j, s_{j+1}, ..., s_{j+k}\} \text{ more than any other } l \in L \\ no & \text{otherwise} \end{cases}$$

As stated earlier selecting the label that most closely resembles a given sequence is up to the implementation. This selection may not always be easy to implement, and is what per our definition quickly separates good event sequence predictors from bad event sequence predictors. For the "house" predictive text example it might be easy to have the predictor select the "house" label for a predicted "house" letter sequence, but it may prove more difficult to determine if the chess moves {"Pawn E2 to E3", "Rook D2 to D4", ...} resemble the "Sicilian Defense" label more than the "Spanish Opening" label.

Whenever S may potentially resemble multiple candidate labels $L_c \subseteq L$ equally well, the event sequence predictor must still select only one label $l \in L_c$. Which label to select is up to the prediction method employed, but one way of resolving this conflict is to select the candidate label $l \in L_c$ that has appeared most frequently during predictor training.

Event sequence prediction can also be made probabilistic instead of deterministic. If so, instead of returning a single label $l \in L$ for the predicted sequence we calculate a distribution $p(l|\{s_i, ..., s_j, s_{j+1}, ..., s_{j+k}\})$ for each label $l \in L$ describing the probability of the predicted sequence resembling label l. These probabilities depend on how much each label resembles the given sequence, which is again up to the implementation just as in the deterministic case of selecting only one label $l \in L$.

Having these distributions for the labels in L can be useful for instance when wanting to list the most probable words when a user is typing letters within a predictive text system, rather than only the single most probable word.

2.6 Summary

We have now defined events and event sequences, along with general sequence prediction and recognition. We have from this created definitions for labeled event sequences and event sequence prediction. For all prediction and recognition definitions we have also briefly described how they can be made probabilistic rather than deterministic. In the next chapter we make use of these definitions when describing two existing approaches to sequence prediction.

CHAPTER

THREE

RELATED TECHNIQUES FOR PREDICTION

In this chapter we will discuss two different methods for predicting sequences. We will look at classification trees, a prediction method that has been used in many areas among them data mining and machine intelligence, and also n-grams as they have been used to predict natural language, which is our test domain. These methods will be used as comparisons to sequence trees when they are tested in Chapter 7.

3.1 Classification Trees

In this section we will look at a classification method called classification trees; in data mining literature it is also known as decision trees as described by Mitchell[14]. The method is a widely used type of classifier that uses a tree structure to classify, where internal nodes are tests leading to labels in leaf nodes that classify a given data instance.

Some of the strengths of classification trees is that their underlying model of the data is easy to visualize and interpret, and the classifier is able to handle both discrete and numeric data without requiring the trained data to be preprocessed or be presented in a specific format.

Classification trees classify data by walking from root node to leaf nodes. At each node a predictor variable will be evaluated and the next edge to follow defined by the outcome of the evaluation, leaf nodes are all labeled with a class label and a data instance is classified as the label of the reached leaf node.

The intuition behind classification trees is to have the most defining attributes near the root of the tree, such that in principle only a few variables may be needed to classify a data instance, making classification quite fast in some cases.

The classification tree learning algorithms we will discuss are divide and conquer algorithms using greedy search that uses information gain as a heuristic to choose good tree structures. We will first discuss the basic learning algorithm ID3 that forms the core mechanics of the later C4.5 algorithm that will be used for testing in Chapter 7.

The definition of classification trees that we use is from the lecture notes of Manfred Jaeger[8], where $sp(a)$ is the state space of predictor attribute a.

Definition 11. *Classification Tree*

A classification tree T is a classification model for a class variable c given attributes A where:

- *Internal nodes are labeled with predictor attributes $a \in A$;*
- *The outgoing edges of an internal node for attribute a are labeled with (subsets of) the states in $sp(a)$, such that each state of a appears in exactly one label.*
- *Each leaf is labeled with a state from $sp(c)$.*

To create a classification tree consider the data presented in Table 3.1. This table contains a sample dataset of observations of weather, temperature and the number of participants, and whether or not they play football. These observations can be used for determining whether or not it is time to play football in similar situations. Each row in the table is an instance of the situation which is to be classified. Each column is a predictor variable or class variable and its state in a given instance.

A dataset for a classification tree is said to be noise free if there is not two identical instances with differing class variable values. Classification trees can be used to classify datasets with missing attributes, but we will not go into details on this subject. Both of these situations obviously lessens the chance to classify the class variable correctly.

	Predictor attributes			Class variable
Instance	Weather	Temperature	Participants	Playing Football
1	Sunny	Warm°	Many	Yes
2	Rainy	Cold°	Many	No
3	Rainy	Warm°	Few	No
4	Cloudy	Cold°	Many	No
5	Cloudy	Warm°	Many	yes
⋮	⋮	⋮	⋮	⋮

Table 3.1: A sample dataset for a classification tree to determine whether or not someone is playing football.

3.1.1 Classification Tree Learning Basics

To create a classification tree the first step is to find the attribute that will give the most information. This is calculated using the information gain, the expected reduction of entropy caused by knowing the value of the attribute. To calculate this the entropy for each attribute must first be found. Entropy is used as a measure of information impurity. Given a probability distribution P we can calculate its entropy as described by Michell[14]:

$$Entropy(P) = -\sum_{i=1}^{n} p_i \cdot \log_2(p_i) \tag{3.1}$$

The information gain for a set S of data instances and an attribute a can then be calculated as shown in Equation 3.2, where $S_v \subseteq S$ is the subset of S with attribute value v.

$$Gain(S, a) = Entropy(S) - \sum_{v \in Values(a)} \frac{|S_v|}{|S|} Entropy(S_v) \tag{3.2}$$

For each attribute the information gain can be computed and the attribute with the highest information gain will be chosen as the current predictor attribute. If we use the data instances from Table 3.1 denoted as $S_{Footbal}$, we can calculate the information gain for each attribute using the formula from Definition 3.2. We will show the information gain calculation for the weather attribute. First we insert the weather attribute in the information gain equation, shown in in equation 3.3.

$$\begin{aligned} Gain(S_{Footbal}, Weather) = Entropy(S_{Footbal}) - (\tfrac{2}{5} \times Entropy(Weather_{Cloudy}) \\ + \tfrac{2}{5} \times Entropy(Weather_{Rainy}) \\ + \tfrac{1}{5} \times Entropy(Weather_{Sunny}) \end{aligned} \tag{3.3}$$

We need to calculate the entropy of the weather attribute of the whole set of data instances shown in table 3.1. In equation 3.4 we consider the probability distribution of the class variable's value in the weather attribute.

$$\begin{aligned} Entropy(S_{Footbal}) &= -(\tfrac{2}{5} \times \log_2(\tfrac{2}{5}) + \tfrac{3}{5} \times \log_2(\tfrac{3}{5})) \\ Entropy(S_{Footbal}) &= -(\tfrac{2}{5} \times -1.32) + \tfrac{3}{5} \times -0.74)) \\ Entropy(S_{Footbal}) &= 0.97 \end{aligned} \tag{3.4}$$

The next step is to calculate, the entropy of the different values in the weather attribute. The first is the rainy value of the weather attribute. The calculation is shown in equation 3.5. For all cases of rainy weather the class variable is *no* thus the entropy will be zero.

$$\begin{aligned} Entropy(Weather_{Rainy}) &= -(0 \times log_2(0) + 1 \times \log_2(1)) \\ Entropy(Weather_{Rainy}) &= -(1 \times 0) \\ Entropy(Weather_{Rainy}) &= 0 \end{aligned} \tag{3.5}$$

The next value to calculate, in equation 3.6 is cloudy weather, in this case we have an equal distribution between playing and not playing football, and thus we will have an entropy of 1.

$$\begin{aligned} Entropy(Weather_{Cloudy}) &= -(\tfrac{1}{2} \times \log_2(\tfrac{1}{2}) + \tfrac{1}{2} \times \log_2(\tfrac{1}{2})) \\ Entropy(Weather_{Cloudy}) &= -(\tfrac{1}{2} \times -1 + \tfrac{1}{2} \times -1) \\ Entropy(Weather_{Cloudy}) &= 1 \end{aligned} \tag{3.6}$$

The final value of weather is sunny, calculated in equation 3.7, as with rainy all data instances with sunny has the same class variable value and we again get an entropy value of 0.

$$\begin{aligned} Entropy(Weather_{Sunny}) &= -(0 \times log_2(0) + 1 \times \log_2(1)) \\ Entropy(Weather_{Sunny}) &= -(1 \times 0) \\ Entropy(Weather_{Sunny}) &= 0 \end{aligned} \tag{3.7}$$

Combining all these results into the information gain equation we get an information gain of 0.57, as shown in equation 3.8.

$$Gain(S_{Footboll}, Weather) = 0.97 - (\frac{2}{5} \times 0) + \frac{2}{5} \times 1) + \frac{1}{5} \times 0)$$
$$Gain(S_{Footboll}, Weather) = 0.57 \tag{3.8}$$

The information gain from weather and the two other predictor variables can be seen in Table 3.2

Predictor variable	Information Gain
Weather	0.57
Temperature	0.17
Participants	0.42

Table 3.2: The information gain for each predictor attribute when finding a candidate for the root node from the data in table 3.1

The values in Table 3.2 show that the most gain is from evaluating the weather attribute first, this will be used as root node. For each value that the weather attribute can attain an edge is created, for both sunny and rain these instances only lead to a single classification thus a leaf node is appended with the classification. For the edge with the cloudy value the process of calculating information gain must be done again, on the remaining predictor attributes and the data instances that has a weather attribute value of cloudy. To conclude the example and the classification tree learning we next select temperature as it classifies the remaining data instances completely and then tree learning is complete. The resulting classification tree can be seen in Figure 3.1.

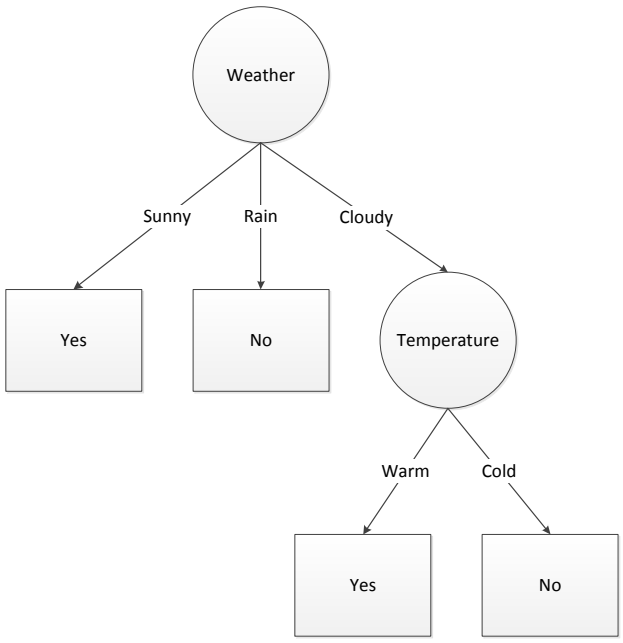

Figure 3.1: A classification tree for the data instances in Table 3.2

When the classification tree is finally constructed we can use it to classify new data instances, one of these is shown in Table 3.3

	Predictor attributes		
Instance	Weather	Temperature	Participants
1	Cloudy	Warm	Many

Table 3.3: An unclassified data instance for the football example

To classify this new data instance consider the classification tree in figure 3.1. The root node is weather, thus we will look at this variable first, the value is cloudy and go along the cloudy edge. As we have not reached a leaf node we need to consider another predictor variable temperature, the value of this, in the data instance is warm, using the corresponding edge we reach a leaf node with the label Yes, which is the classification of the new data instance.

A more advanced method of classification tree learning is the algorithm C4.5 a descendant of ID3. An implementation of the C4.5 algorithm called J48 will be used as a basis for the classification tree learning used in the tests described in Chapter 7. In the following sections we will describe the C4.5 algorithm.

3.1.2 Classification Tree Learning via C4.5

The C4.5 algorithm has several improvements over the basic classification tree algorithm described above. The algorithm uses a more complex heuristic for its greedy search called the splitting criterion, which takes account for missing data and improves on predictor attributes with many possible values, as this poses an issue in the previous method. Additionally to reduce the danger of overfitting, and to reduce the tree size the algorithm includes a post-pruning technique. We describe the C4.5 algorithm explained by Kohavi & Quinlan[10].

C4.5 Classification Tree Learning

The C4.5 algorithm works in two steps, first the classification tree is built, and then the tree is pruned to reduce the tree size while maintaining an acceptable classification accuracy.

As before we initially consider the full set of data instances, used as training cases, defined as S. An individual data instance $s_j \in S$ is denoted as $s_j = (a_0, ..., a_k, c_j)$, where a_k is the value of attribute k and c_j is the value of the class variable for instance s_j.

In the C4.5 algorithm there are a number of different tests that can be done on the values of predictor attributes. These tests, defined as $B = (b_1, ..., b_t)$, partition the cases in the training set S into t parts.

The algorithm considers at each step a subtree $T_i \in T$, where T is the full tree, created from the partition S_i of the training set S. The root node is T_0 with a partition $S_0 = S$.

At each step the C4.5 algorithm creates the subtree T_i. At this creation step there are two options based on the current training set S_i:

1. T_i is created as a leaf node labeled as c_j if all the training cases in S_i have the same class value c_j, or

Temperature	14°	16°	29°	32°	35°
Go Outside	no	no	no	yes	yes

Table 3.4: Attribute discretization example; the threshold is 32° celcius.

2. T_i is created as a classification subtree, created from a new partition of S according to some new test B.

In the first case with a leaf C4.5 only needs to label the new node with the class value. In the second case a candidate test must be found that best divides the training set S based on one attribute a of the attributes in the training instances.

Candidate Test Types

As mentioned before the C4.5 algorithm has different types of candidate tests that can be used based on the type of the values in attribute a:

1. If a is a discrete attribute with z values, one of two test types exists in C4.5:
   (a) The first is a "$a = ?$", or "a has what discreet value?" test. This test has z outcomes, one for each attainable value of a. This is the same as described in the basic learning method described in section 3.1.1.
   (b) The second test is a "$a \in G_?$", or "a is in which grouping?" test with $2 \leq g \leq z$ outcomes, where $G = \{G_1, ..., G_g\}$ is some partition of the values of a. Choosing the best partition G for this test relies on the splitting criterion. Using the data instances from Table 3.1 and the predictor variable weather we can group cloudy and rainy into a single group called bad weather, using this we will end up with only two discrete values of the weather predictor attribute.
2. Lastly if a is a numeric attribute, another type of test exists in C4.5:
   - a "$a \leq \theta$", or "is a less than or equal to θ" test with a discrete binary *true/false* outcome, where θ is some constant threshold. Possible thresholds are found by sorting the known values of a, and identifying where the class attribute value changes between any two adjacent values of a as exemplified in Table 3.4. Choosing the best of these thresholds relies on the splitting criterion to be discussed below.

Splitting Criterion

At each subtree several test candidates can arise, as with the basic tree learning method we must find and choose the predictor variable where we gain the most information. However in C4.5, we must not only find the highest information gain, we may also have to find the best grouping G of discrete variables and the best threshold θ for numeric values. Choosing the best of these candidates, as well as choosing an attribute value partition G and threshold θ, relies on the splitting criterion. What we wish to do is to find the best test candidate, and to do this it may be necessary

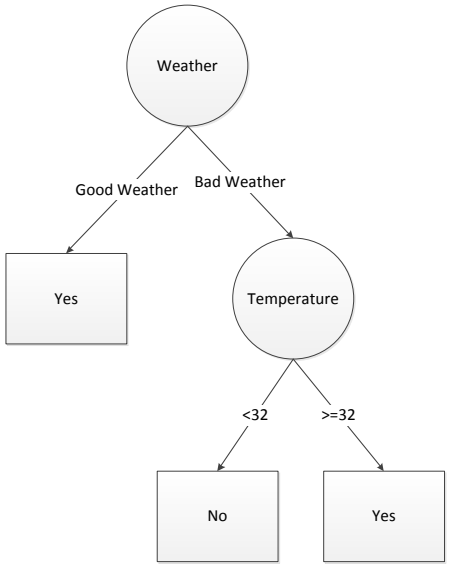

Figure 3.2: A sample classification tree that may be constructed using the c4.5 algorithm

to find the partition G or threshold θ that maximize the splitting criterion. To find the best candidate test, a greedy search for the candidate maximizing the splitting criterion heuristic is performed.

The splitting criterion is based on the information gain as defined in Definition 3.2 but uses a more complex equation, as the original method used to calculate information gain favors splits that have many outcomes. Another improvement to the calculation of information gain is to consider missing values during training.

When the best candidate test has been found the current subtree T_i will be labeled with the candidate test predictor attribute and an edge for each possible outcome of the candidate test. In the case of our grouping of the weather predictor attribute's values into good weather and bad weather we would create two edges and have to consider two new subtrees.

The training instances S_0 is then split into smaller sets for each new subtree, where the instances corresponds to the test outcome of the edges leading to the new subtrees. In our example we have one set with all cases where the weather predictor attribute is good weather and another set where it is bad weather.

The algorithm will then evaluate the next subtrees, created from the test results of the weather test, until there are created no new subtrees, i.e. we arrive at the fist option, that the class variable has the same value for each data instance in S_i. A sample classification tree where the grouping and discretization we have discussed above may look as depicted in figure 3.2

When the learning of the tree is finished the classification tree may be very large and over fitted to the training data, the C4.5 algorithm therefore incorporates as a final step a pruning technique that we will now discuss.

Classification Tree Pruning

Post pruning is used both to reduce the classification tree size, but also to avoid overfitting the classifier on the training data. If all instances of the training data are predicted correctly, there may be a risk of the classifier having lost its ability to generalize. In this case the classifier is likely to drop in accuracy when predicting on untrained data. The C4.5 algorithm makes use of post pruning after the full classification tree has been constructed. This post pruning relies on estimating the true error rate of the classifier, which will be explained in the following.

This post pruning is done by evaluating each subtree T in the classification tree in a bottom-up pass. Such a subtree T can be seen in Figure 3.3 as explained by Kohavi & Quinlan[10].

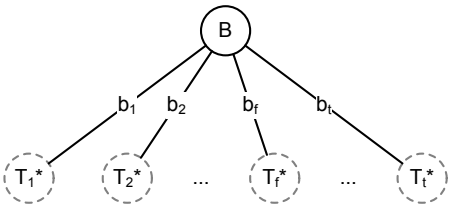

Figure 3.3: Example subtree T for C4.5 classification tree post pruning

In that figure, B is the test performed for the currently evaluated subtree T with outcomes $b_1, ..., b_t$, and $T_1^*, ..., T_t^*$ are other already pruned subtrees. The subtree can be modified in three different ways, what modification that is chosen is based on which of the three gives the least error in classification. The three options are:

- Do not prune subtree T
- Replace subtree T with the label L with the most frequent class in T
- Replace subtree T with subtree T_f^* that is the most frequent outcome of test B

When bottom-up post pruning has completed for the root node of the classification tree there are no more subtrees to prune and the C4.5 pruning algorithm is done. The classification tree has now possibly been both reduced in size and made more general, reducing overfitting and allowing the classifier to better predict on untrained data.

One should note that since this is a form of post pruning the classification tree can grow quite large before pruning is begun depending on the training data, and this unpruned classification tree may require substantial amounts of storage compared to that required by other prediction methods.

3.1.3 Using Classification Trees to do Event Sequence Prediction

To use classification trees for event sequence prediction sequences are encoded as a number of predictor attributes $a_1, ..., a_n$ where n is the number of events in the largest

known sequence. Each predictor attribute corresponds to the event at position i in the sequence, such that $a_i = s_i$. If a sequence during training is shorter than n, the remaining predictor attributes will contain a blank data symbol. During prediction these predictor attributes contain the '?' symbol to describe missing information instead.

This data structure may not be optimal for classification trees as there will be a large number of predictor attributes with a sizable number of values, up to the alphabet size in our domain, but is the best way to have a similar way of using each event in the sequence as is done with sequence trees. It is also this data structure that requires a more complex classification tree algorithm as there exists many cases of unknown values in the data instances, which C4.5 takes into account when constructing the tree.

Sequence labels fit neatly into the definition of the class variable used in classification trees, each instance has a single class variable with a value, which we use as the label.

To do sequence prediction using classification trees we construct the tree using the structure mentioned above. This results in a tree structure ordered by the most defining position in the sequence. A sample tree can be seen in Figure 3.4, here the most influential position in a sequence is position 2, where we have two possible values, a and b.

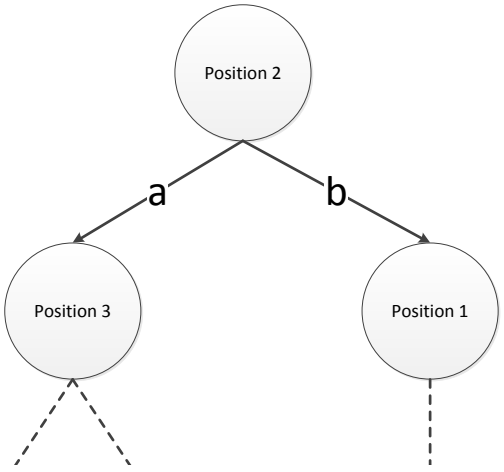

Figure 3.4: A possible Classification tree made using sequences

When a tree is built a sequence can be predicted by inputting a data instance with a number of events such as $\{a, b, ?, ?\}$ as the subsequence of $\{a, b, c, d\}$. The classification then as per normal does its classification on the predictor attributes, and returns a label.

Having discussed classification trees and the C4.5 algorithm classification trees are based on we now move on to describe the second prediction method we will use as benchmark for our sequence tree prediction method n-gram predictor.

3.2 N-grams

N-grams are a method of inferring correlations between elements in a sequence. N-grams have been used in various statistical analysis techniques and are not limited to prediction. They have also been used with success in the context of natural language processing.

Within language processing n-grams have been used to suggest alternatives to misspelled words, they have been used to identify which language a text is written in, and even to generate sentences on the fly in the "Dissociated Press" GNU Emacs text editor command as described by Stallman[16].

N-grams use the n prior elements in a sequence to determine the probability of the next element, similar to how sequence prediction is described in Definition 5 of Chapter 2. It is in effect a n-order Markov model. The probability of the next element occurring is calculated in equation 3.9.

$$P(s_{j+1}|s_j, ..., s_{j-n+1}) \tag{3.9}$$

Strictly, an n-gram algorithm keeps track of the frequency of each sequence rather than the probability. An example 3-gram will keep track of the number of times each sequence consisting of 3 elements is seen. For prediction the sequence would give a distribution of all the possible options after the sequence.

By using frequency instead of probability one can optimize the data structure used to store N-grams. Frequencies also have the advantage of being easier to update.

In our discussions we will use Definition 12 to describe the n-gram prediction method:

Definition 12. *N-gram prediction method*
An n-gram predictor contains a list of n-grams which are triples consisting of the following elements

$$(\{s_{j-n+1}, ..., s_j\}, s_{j+1}, \omega)$$

Where

- $\{s_{j-n+1}, ..., s_j\}$ *are the n previous events in a sequence,*
- s_{j+1} *is the prediction, and*
- ω *is the frequency of this prediction occurring.*

As can be seen n-grams are three-tuples consisting of the previous n events, the predicted event and the frequency of that prediction occurring when the previous events have been observed. An n-gram predictor is then a list of such n-grams.

With this definition we can use n-grams as a prediction method. There are two steps we must do, firstly we must find n-grams in training data. Consider the sentence:

Ich bin ein Berliner

Using a 3-gram model this sentence would be divided into:

3-grams	{({?,?,?},Ich,1), ({?,?,Ich},bin,1), ({?,Ich,bin},ein,1), ({Ich,bin,ein},Berliner,1), ({bin,ein,Berliner},∅,1)}

When we have found a number of n-grams to predict with we can then look at other sequences and consider the n last events of the sequence and use the learned n-grams to suggest a prediction. A new sequence could be "Ich" we consult the learned n-grams and we can convert this sequence to a sequence and find it among the learned n-grams and see what is the most probable next event. in this case we will ofcouse create the sequence $\{?, ?, Ich\}$, looking among the learned n-grams we can see that there can only be one outcome namely the event "bin". This is one of the simplest methods to predict using n-grams but a way of improving prediction accuracy for n-grams when frequencies are low is to use hierarchical n-grams.

3.2.1 Hierarchical N-grams

Hierarchical N-grams effectively have several N-grams working in parallel with an increasingly larger window size. For example a hierarchical 3-gram will have a 1-gram, 2-gram and 3-gram processing the same data. When constructing a hierarchical N-gram from a sequence it is registered in all the N-grams.

Again consider the string:

Ich bin ein Berliner

A hierarchical 3-gram model would divide the string into the following n-grams:

3-grams	{({?,?,?},Ich,1), ({?,?,Ich},bin,1), ({?,Ich,bin},ein,1), ({Ich,bin,ein},Berliner,1), ({bin,ein,Berliner},∅,1)}
2-grams	{({?,?},Ich,1), ({?,Ich},bin,1), ({Ich,bin},ein,1), ({bin,ein},Berliner,1), ({ein,Berliner},∅,1)}
1-grams	{({?},Ich,1), ({Ich},bin,1), ({bin},ein,1), ({ein},Berliner,1), ({Berliner},∅,1)}

Table 3.5: Hierarchical n-grams example

When predicting on a hierarchical N-gram you first look up the prediction for the n'th gram. If there have been sufficient examples of the n'th gram it uses this prediction. If this is not the case n-gram n-1 is examined following the same criteria. If none of the n-grams have sufficient examples a random prediction can be returned.

What constitutes as sufficient of course depends on the application. There is no single correct threshold value for the number of entries required for making a confident prediction. It is to some extent a matter of trial and error, but it can be common to simply use a small threshold value of 3 or 4 occurrences.

3.2.2 Open-loop N-gram Predictions

An issue with n-grams is that it predicts a single next element. In the previous example it was only possible to infer the next element, but for predictive text we wish to predict sequences of letters, based on a sequence of previous letters. We can accomplish this by using recursive n-grams to do open-loop prediction.

Recall that we in Section 2.2 mentioned open-loop prediction; recursive n-gram prediction can be seen as a case of this. Instead of predicting only the next event in the sequence, recursive n-grams predict a whole sequence of events by basing its next prediction on a previous prediction:

$$\begin{aligned} \{s_{j-n+1}, ..., s_j\} &\rightarrow s_{j+1} \\ \{s_{j-n+2}, ..., s_{j+1}\} &\rightarrow s_{j+2} \\ &... \\ \{s_{j-n+k}, ..., s_{j+k-1}\} &\rightarrow s_{j+k} \end{aligned}$$

All steps included this amounts to what was shown in Definition 5 for sequence prediction, where $1 \leq i \leq j \leq \infty$ and $k \geq 1$:

$$\{s_i, ..., s_j\} \rightarrow \{s_{j+1}, ..., s_{j+k}\}$$

In both situations previous events, either $\{s_{j-n+1}, ..., s_{j+k-1}\}$ or $\{s_i, ..., s_j\}$, are used as a basis for predicting the next events $\{s_{j+1}, ..., s_{j+k}\}$. Also in both situations the natural number $k \geq 1$ dictates when to stop prediction. For predictive text this can be when a full word has been produced.

We determine when a full word has been produced when the next n-gram prediction is a terminal symbol. In our implementation the terminal symbol is whitespace, as this is a natural terminator for words. If the letter t is given as input, our n-gram predictor will return the following:

Step	Output	Found n-gram
1	"t"	$(\{?, ?, t\}, h, 1)$
2	"th"	$(\{?, t, h\}, i, 1)$
3	"thi"	$(\{t, h, i\}, s, 1)$
4	"$this$"	$(\{h, i, s\}, \_, 1)$
5	"$this\_$"	

When a whitespace is predicted we terminate the prediction process. This enables us to predict full words. Additionally, as this prediction process might produce an infinite string in some cases, we terminate the process after a maximum number of steps to avoid infinite recursion.

Next we describe in more detail how we can use such recursive n-grams to do event sequence prediction within the predictive text domain. This includes how to translate predicted letter sequences into word labels.

3.2.3 Using N-grams to do Event Sequence Prediction

In this section we describe n-grams as we use them for text prediction. In this domain we use 3-grams to predict a word label based on a sequence of letters, similar to how we earlier in this chapter used classification trees to predict word labels based on letter sequences.

To do word prediction with n-grams we use the recursive n-grams method described above, and once a terminal whitespace symbol _ is predicted as the next symbol the complete string that has been constructed so far is returned as a word label.

An example is to have the trigram method predict the word "house" based on the letters "ho" and the previously observed words "house" and "hound" with frequency 10 and 5 respectively. The 3-grams resulting from training are as follows:

$$
\begin{array}{lll}
(\{?,?,?\},h,15) & (\{?,?,h\},o,15) & (\{?,h,o\},u,15) \\
(\{h,o,u\},s,10) & (\{h,o,u\},n,5) & (\{o,u,s\},e,10) \\
(\{o,u,n\},d,5) & (\{u,s,e\},\_,10) & (\{u,n,d\},\_,5)
\end{array}
$$

As per recursive n-gram prediction the most likely successor of "ho" is the 3-gram $(\{?,h,o\},u,15)$, and following this the most likely successor of "hou" is $(\{h,o,u\},s,10)$, followed by $(\{o,u,s\},e,10)$ and $(\{u,s,e\},\_,10)$. This produces the word "house" as was originally intended.

When a terminal symbol is predicted a translation from the predicted sequence to a word label is done using a dictionary lookup. This dictionary provides a mapping from a predicted sequence to a word label, and each lookup is done as an additional step after prediction has finished.

This makes the n-gram predictor only predict labels rather than generating new previously unseen words, although generation of new bogus words is quite possible. It was an amusing feature of the n-gram predictor to have it generate nonsensical words during early prediction testing, but for the sake of method comparison we ultimately had to conform it to producing labels similar to the classification trees method and sequence trees explained in the coming chapter.

3.3 Summary

The two methods classification trees and n-gram prediction that we will use as benchmarks for our own sequence prediction method have now been described. For classification trees we will use the J48 algorithm in Weka which is an open source

implementation of the C4.5 algorithm described in the chapter. In the case of n-grams we will use open-loop prediction via recursing n-grams and then convert these to labels such that we will have output from n-grams that is comparable to that of the other two predictors in the test chapter later. We will now proceed to introduce our own prediction method called sequence trees, specifically the tree construction algorithm and several optimizations for this process.

CHAPTER

FOUR

SEQUENCE TREE CONSTRUCTION

In this chapter we will define sequence trees, as well as the algorithm for sequence tree construction. We introduce the construction algorithm and discuss four optimizations to this construction method that we will test in Chapter 7. The optimizations we propose are to collapse nodes in the sequence tree to losslessly compress the size of the sequence tree, the second is to use tree pruning to create an approximate tree with lower size, finally we propose to cluster the training data and select candidates to represent clustered sequences thus reducing the training data.

4.1 Sequence Trees

Sequence trees are a sequence prediction method that uses supervised learning. With sequence trees we attempt to define a generalized sequence prediction method. The idea behind sequence trees is to use trees to define a simple representation of recognizable sequences and their predictions. The training and prediction time is quick, but the method has an issue with a large spatial requirements, so we will later introduce suggestions for reducing these requirements.

In sequence trees a path from root to leaf node corresponds to a sequence of events found in the modeled system. The root node represents the event from which all initial events occur; this node contains a special event ϕ signifying no event. Internal nodes corresponds to events possible to occur given their parents. Each leaf node in the sequence tree has a label node appended, these labels define the classification of the paths leading to the leaf node.

We define sequence trees in Definition 13 below.

Definition 13. *Sequence Trees*
A sequence tree is a directed acyclic graph (N, L, E) *where*

- N*, is a finite set of nodes, each representing an event.*
- $N_\phi \in N$ *is the root node, containing the empty symbol* ϕ*.*
- L*, is a set of labels appended to the leaf nodes of the sequence tree.*

- *E, is a finite set of directed edges defined as $(N \times (N \cup L), \mathbb{N})$, representing the node, its child node and the frequency of seeing the child node given its ancestor nodes.*

Sequence trees follow a standard tree structure with leaves at the end of each branch. In our case these leaves are used as labels describing the sequence of events leading to them. Edges connect nodes to other nodes or labels and are annotated with the frequency of the next event occurring when all the previous events have occurred.

To give an example of such a sequence tree, consider the sequence tree in Figure 4.1. The nodes of the tree are defined as $N = \{\phi, A, B, C\}$, which gives us three internal nodes and the root node. The labels of the tree are $L = \{1, 2\}$. Lastly the edges to connect them all are $E = \{(\phi \rightarrow A, 3), (A \rightarrow B, 2), (A \rightarrow C, 1), (B \rightarrow 1, 2), (C \rightarrow 2, 1)\}$.

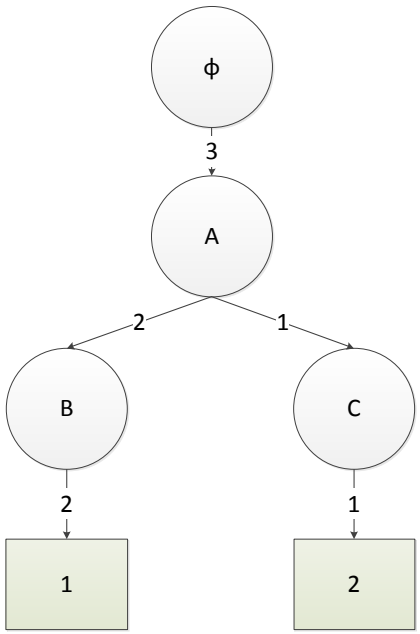

Figure 4.1: A small sequence tree

More advanced examples of sequence trees and their use for prediction will be given later. With the general definitions complete we will now introduce sequence tree sequence insertion that is an essential part of the construction of sequence trees.

4.2 Sequence Tree Insertion

For most tree-based data structures data insertion is one of the most common operations. Sequence insertion is no different and is a basic operation on sequence trees that is used extensively. One difference remains, however, as sequence insertion for sequence trees takes an entire sequence of events as input rather than single events, as is common for other tree-based data structures.

Using such event sequences rather than single events allows sequence tree insertion to account for the ordering of sequence events when updating the tree structure; insertion of single events actually makes little sense when concerned with sequences, as use of the ordering of events is an essential part of sequence trees.

Insertion of an event sequence can be seen described in Algorithm 1. The new sequence is inserted as a path in the tree, one event at a time, with previously missing events being added as new nodes with connected edges when necessary.

Algorithm 1: Sequence Tree Insertion Algorithm

Input: A sequence tree T, a labeled event sequence s
Output: The updated sequence tree T

```
1 currentNode ← root ∈ T
2 foreach event and label x ∈ s do
3     edge ← Find edge leading to x ∈ currentNode
4     if edge does not exist then
5         edge ← Create new edge
6         edge.child ← Create new node containing x
7     Increment frequency of edge
8     currentNode ← Node pointed to by edge
9 return T
```

The insertion algorithm is very basic in its method of adding data to the model. It simply iterates through the labeled event sequence and adds new nodes when needed while continuously updating the frequencies in the tree. This simplicity comes at the cost of relatively large spatial requirements as we are not making any inference on the data given; the data is simply stored in the sequence tree data structure. This however makes for a very quick insertion speed.

The quick insertion speed can be seen by the time complexity of the algorithm that is only $O(l)$, where l is the maximum length of a sequence. At the point of insertion the algorithm makes no effort to do any reduction or inference on the data; instead we keep this separate and we will later show several methods that perform operations on the tree to reduce size, as well as to improve prediction speed and accuracy.

With insertion of a sequence into a sequence tree covered we move on to sequence tree construction that makes heavy use of this insertion.

4.3 Sequence Tree Construction

To construct a sequence tree from sequential data the data is first converted into labeled event sequences and then given as input to Algorithm 2 which describes sequence tree construction as a series of labeled event sequence insertions. These labeled event sequences are thus the training set for the sequence trees classifier.

What follows is an example using the construction and insertion algorithms to construct a sequence tree. Initially the sequence tree only consists of the root node containing ϕ. Using Figure 4.2 as an example we will construct a sequence tree

Algorithm 2: Sequence Tree Construction Algorithm

Input: A set of labeled event sequences S
Output: A constructed sequence tree T

```
1 T ← Create empty sequence tree
2 Create empty rootNode ∈ T
3 foreach labeled sequence s ∈ S do
4 |   T ← InsertionAlgorithm(T, s)
5 return T
```

consisting of the labeled event sequences $(\{A\}, 1)$, $(\{B, B\}, 2)$ and $(\{B, A\}, 3)$ as our dataset.

To construct the sequence tree we start in the root node, and consider the first sequence ({A},1). We check whether the event in the sequence exists as a child node. If it does we walk down the edge towards it. If it does not exist such a node is created and connected to the current node, and we walk towards the new node. As the event A does not exist in the tree we create a new node containing it and give the edge leading to it a frequency of 1. This process is repeated in the child node, where it checks whether the next event is present in the tree, and create it if it is not. This is continued until the end of the sequence for each event and label in the labeled event sequence. With $(\{A\}, 1)$ we have processed the whole sequence and only need to add the label 1, this is done in the same almost the same manner as before with the creation of a new node where we now append the node with the label instead of an event.

The process is then repeated for the next labeled event sequence $(\{B, B\}, 2)$ resulting in the sequence tree as seen in Figure 4.2 step 3. And lastly with the sequence $(\{B, A\}, 3)$ added to the tree, note that we here did not add a new node B to the tree as it already existed, instead the frequency of the edge leading to the existing node was simple incremented.

Once all labeled event sequences from the training data have been inserted the resulting tree T is returned and is ready to be used for prediction. As stated previously sequence tree construction is just a series of insertion operations, and as such sequence tree construction shares its low time complexity.

The time complexity for constructing a sequence tree using Algorithm 2 is $O(|S|l)$ where $|S|$ is the number of sequences and l is the maximum length of a sequence, which is a quite low complexity allowing sequence trees to be trained on large datasets. In the domain of predictive text the alphabet size and sequence length is limited, but the number of sequences can vary a lot given different dictionaries. However, the size of the dictionary is not a factor in the lookup speed. The low time complexity comes at the price of a high space complexity.

The worst case space complexity of sequence trees is when the tree is full; in other words when the tree contains every possible combination of sequences of some finite length l equal to the depth of the tree.

The space complexity of a sequence tree is $O(|N| + |E|)$ where $|N|$ is the number of nodes in the tree and $|E|$ is the number of edges. Edges are included in the calculation as they store information about the frequency of a transition. The maximum

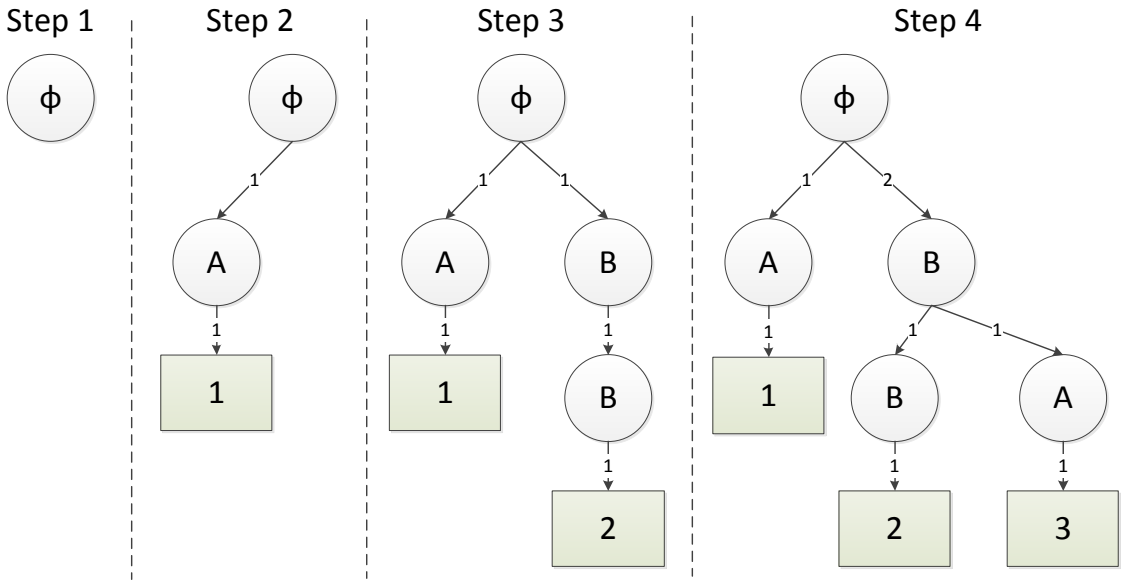

Figure 4.2: Construction of a sequence tree, using the labeled event sequences ({A},1), ({B,B},2) and ({B,A},3).

number of nodes for storing all sequences of finite length l is $|N| = |X|^{l+1} - 1$, and the maximum number of edges is $|E| = |X|^{l+1} - 2$, where X is the alphabet of the event system. Thus in total the space complexity is $O(|X|^l)$

With the basic construction of sequence trees covered we will now turn our attention to methods that can reduce the spatial requirements. First we will cover three different lossless tree collapsing methods to reduce tree size.

4.4 Tree Collapsing - Lossless Tree Size Reduction

In this section we experiment with collapsing of sequence tree nodes to reduce tree size, looking at collapsing of paths, collapsing of subtrees and combining of labels.

In short, path collapsing can be done when a path in the sequence tree has no branches, because once prediction has reached such a path then no matter what input comes after the current the prediction will always provide the same label reached by that one non-branching path.

Collapsing of subtrees is similar and is a more general collapsing method; here entire subtrees of the sequence tree will be collapsed if all paths in such a subtree reach the same label.

For label combining we create only one leaf for each different label and have tree paths point to these rather than having duplicate identical tree leaves such that multiple identical labels do not take up more storage space than necessary. All of these three collapsing methods will now be described.

4.4.1 Collapsing Paths

The first collapsing method we look at is path collapsing, which is a way to reduce the size of sequence trees by collapsing paths where we observe no branching, i.e.

where we have no doubt on the path's resulting label.

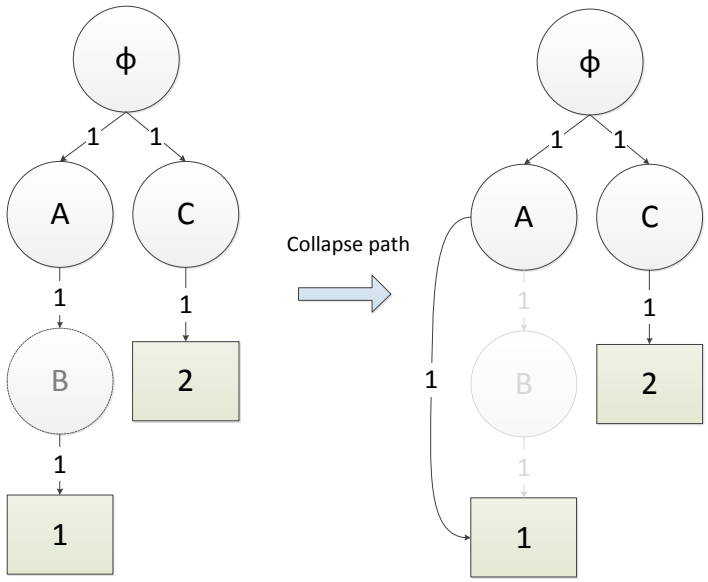

Figure 4.3: An example of path collapsing where the noce B is removed

An example of this can be seen in Figure 4.3. In the figure there is no uncertainty after node A that the resulting label will be 1 in all cases. Therefore it is possible to collapse the path to label 1 bottom-up to the closest branching point in the tree. This means that the node B will be removed and a link from node A to the label will be created instead, which can be seen to the right in the figure.

In essence we use the knowledge that A is a prefix of all event sequences leading to label 1. This method of collapsing paths can also be applied to entire subtrees leading to a single type of label. This optimization can be seen as finding prefixes for event sequences with the same label since, given a set of event sequences, if there there exists a prefix that is unique for a subset of sequences identified by a single label we can substitute the sequences containing the prefix with the prefix itself to identify them.

There is a special case when a path is collapsed internally in the sequence tree. To keep the distance between branches consistent we need to keep a skip value for collapsed nodes. The skip value represents the number of events in the input sequence that must be skipped before being able to continue to a new child node in the tree.

An example of this can be seen for the sequence subtree in Figure 4.4, where node B is removed and its child nodes are connected to parent node A. We store a skip value for each node such that when we collapse internally we will know how many events we need to consume before resuming tree traversal. In the example we give A a skip value of 1, signifying that after reading event A from input the next input event must be consumed before continuing with prediction.

Using these skip values allows for internal path collapsing without losing important information in the sequence tree, while still allowing for space requirement reduction.

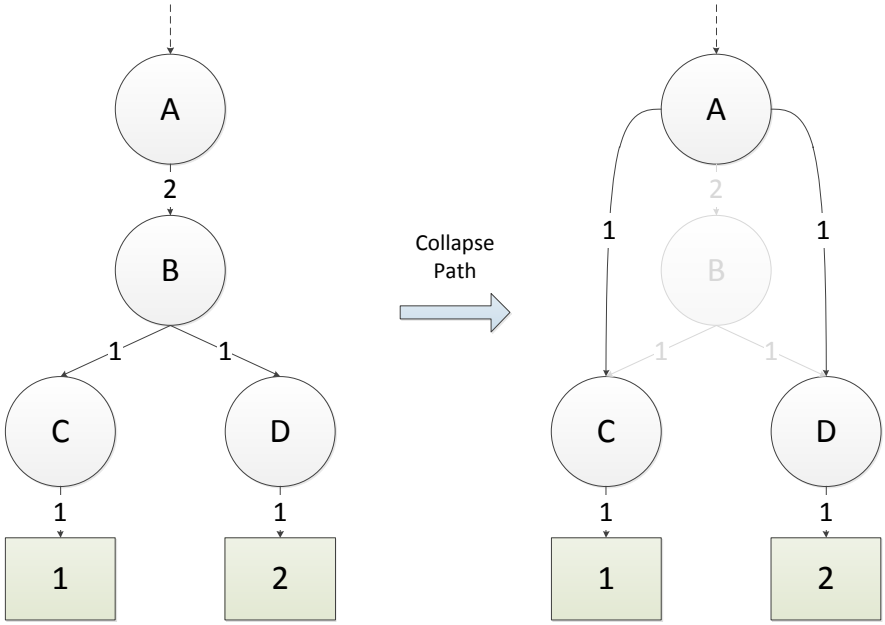

Figure 4.4: Example of internal path collapsing with a skip value inserted in node A after node B has been removed

4.4.2 Collapsing Subtrees

As we saw in Section 4.4.1 we are able to collapse non-branching paths to reduce tree size. We expand on this concept and also look at how it is possible collapse entire subtrees of the sequence tree. To do this the subtree that we wish to collapse must for all paths result in an identical label.

Consider Figure 4.5 as an example. Whenever the first event in a sequence is A then there is no uncertainty in the resulting label prediction, as all possible paths from this point result in a prediction of label 1. Hence it is possible to collapse the subtrees below node A into a single label.

Collapsing these subtrees results in the second sequence tree in Figure 4.5, and a considerable reduction in tree size without any loss of prediction accuracy when predicting labels, as we still retain the original frequencies of the sequence tree; after we have collapsed it the probabilities of reaching label 1 have not been altered.

4.4.3 Combining Labels

One last way we have looked at to reduce tree size is to combine the same labels of different paths into a single combined label. Instead of having duplicate labels we make all paths leading to an identical label refer to a single instance of that label. An example of this combination can be seen in Figure 4.6.

This allows for lossless sequence tree size reduction by making paths with identical labels refer to the same label instead of duplicates, only removing redundant information from the tree. This can potentially result in great size reductions for domains with many sequences leading to only a few labels. In the domain of chess, for instance, many different sequences of chess moves can all lead to the same "Sicilian Defense" opening strategy, and pointing all these sequences to the same label

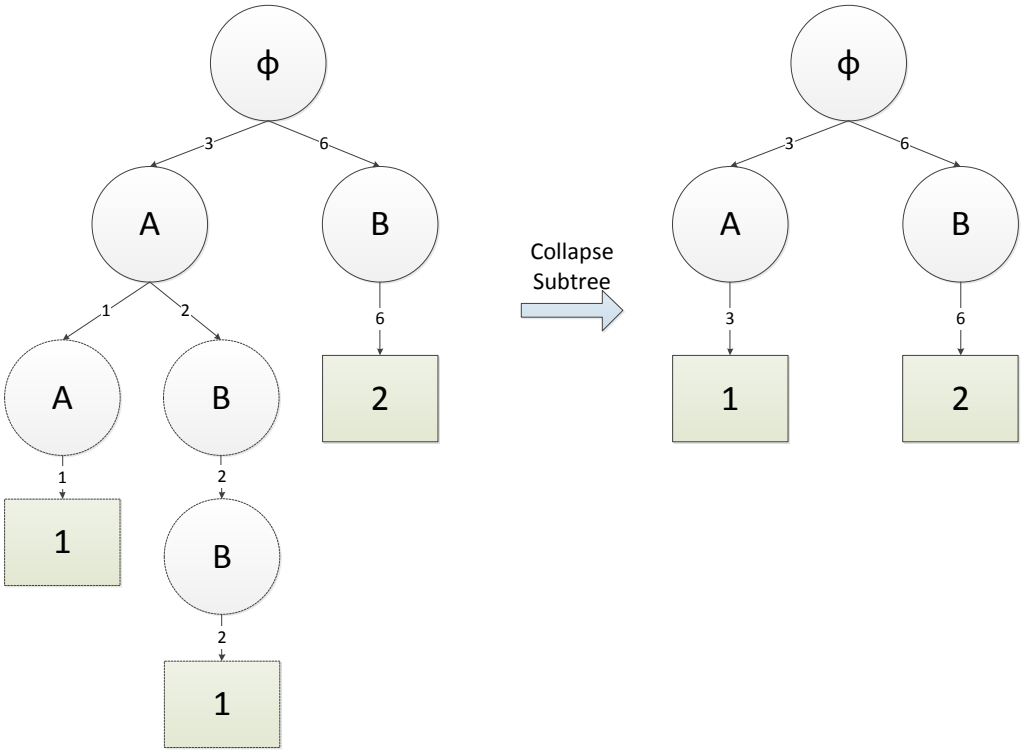

Figure 4.5: Example of a sequence tree where paths leading to the same label are combined into a single label

can result in a great size reduction.

With collapsing paths, collapsing subtrees and combining labels methods covered we will continue on and look at tree pruning as a lossy way to reduce the spatial requirements. However first we need to define subtree removal as this is an essential part of the tree pruning algorithm.

4.5 Sequence Tree Subtree Removal

Apart from sequence insertion, sequence trees also have the possibility to remove data once the tree has been constructed through sequence subtree removal. This is for example used to prune the sequence tree, which we will discuss later in Section 4.6.

Note that this is different from removal of individual sequences from a sequence tree. Sequence removal is not of much use as one could simply have left out the sequences during construction, whereas subtree removal alters the sequences making it useful for more advanced tree manipulations such as those of tree pruning.

The procedure for removing subtrees from a constructed sequence tree can be seen in Algorithm 3.

Nothing happens if the node n to remove is the root node of sequence tree T or does not exist in the sequence tree, as the removal of such a node does not give a valid sequence tree. Otherwise the frequency for node n's parent edge is subtracted from all ancestor edges leading from sequence tree T's root to node n. If any of these

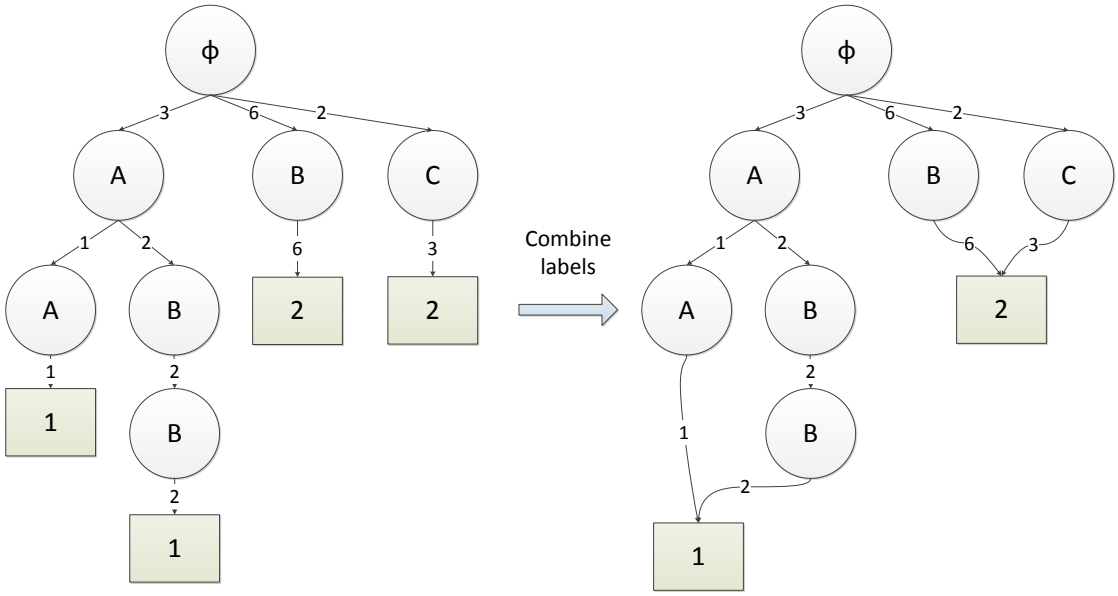

Figure 4.6: Example of a sequence tree where duplicate labels have been combined into a single joint label

edges' frequencies become 0 the edge is removed, resulting in all its child nodes and edges being removed as well.

An example of this can be seen in Figure 4.7 where the node n to be removed is the node B which is the root of sequence subtree $T_{subtree}$ consisting of nodes $N = \{B, C, D\}$ and the labels $L = \{1, 2\}$. As can be seen the frequency of node B's parent edge has been reduced and propagated up through the full sequence tree T after subtree removal has been carried out.

A case to note is when subtree removal causes a path in the sequence tree to no longer have any label assigned. This occurs when a parent has only one child node and this node is removed. In this case all ancestors who lead only to that node are removed from the sequence tree as well.

Figure 4.8 shows an example of this happening where the node n to be removed is B which is the root of $T_{subtree}$ consisting of nodes $N = \{B\}$ and the labels $L = \{1, 2\}$. Step 1 of subtree removal makes the path leading to node B invalid, and therefore node A and its parent edge with frequency 3 are removed in step 2 as well.

As with the insertion algorithm the time complexity of the subtree removal algorithm is $O(l)$ where l is the maximum depth of the sequence tree. This allows for quick altering of the tree for use with pruning algorithms.

With subtree removal covered we can now proceed and discuss tree pruning.

4.6 Tree Pruning - Lossy Tree Size Reduction

Following the three collapsing methods we now look at tree pruning, another way of reducing tree size. Tree pruning, contrary to the lossless collapsing, is however a lossy compression method and will to varying extents result in loss of prediction accuracy. Tree pruning techniques are often used in classification trees to reduce

Algorithm 3: Sequence Tree Subtree Removal

Input: A sequence tree T, root node n of subtree $T_{subtree}$ to remove
Output: A sequence tree T, without $T_{subtree}$

```
if n ∉ T then
    return T
frequencyToReduce ← Frequency of edge pointing to node n
currentNode ← n
while current ≠ RootNode ∈ T do
    parentEdge ← Get parent egde of currentNode
    Reduce frequency of parentEdge with frequencyToReduce
    if Frequency of parentEdge = 0 then
        Remove parentEdge from parent of currentNode's children
    currentNode ← parent of currentNode
return T
```

both the size of the tree and also the chance of it being overfitted to the training data.

The general premise of tree pruning is that if a label or path through the sequence tree is highly unlikely, it may not have a large effect on the accuracy of the sequence tree, and can therefore be removed from the tree. For sequence trees we have investigated three different pruning methods; a brute force pruning method, a low frequency pruning method and a size to frequency ratio pruning method.

4.6.1 Brute Force Pruning

Brute force pruning, the first method to do pruning, follows a method similar to the one used by classification trees in Section 3.1.2; first the initial accuracy of the sequence tree must be measured. Afterwards a test is done for each node in the tree starting bottom-up to try to remove the node, and see whether or not it affects the accuracy of the model. If the removal of the node reduces accuracy by less than a certain threshold, the node can safely be removed and the tree size reduced. However this method is very time consuming and is as its name suggests somewhat of a brute-force method.

4.6.2 Pruning Sequences With Low Frequencies

The second approach is to simply remove sequences with low frequencies in the training data, but the improvement from this is not guaranteed to be a good improvement. If several low frequency sequences share most of their events they would follow the same path in a sequence tree and their probability may not warrant their removal. Consider the sequences "Cowboy" and "Cowboys", "Cowboy" may be a very frequent word whereas "Cowboys" is much less frequent. However as the two words share so many event it would give very a little spatial reduction to remove "Cowboys". Other paths could have nothing in common with other sequences making them more ideal candidates to remove. However the method is incapable of

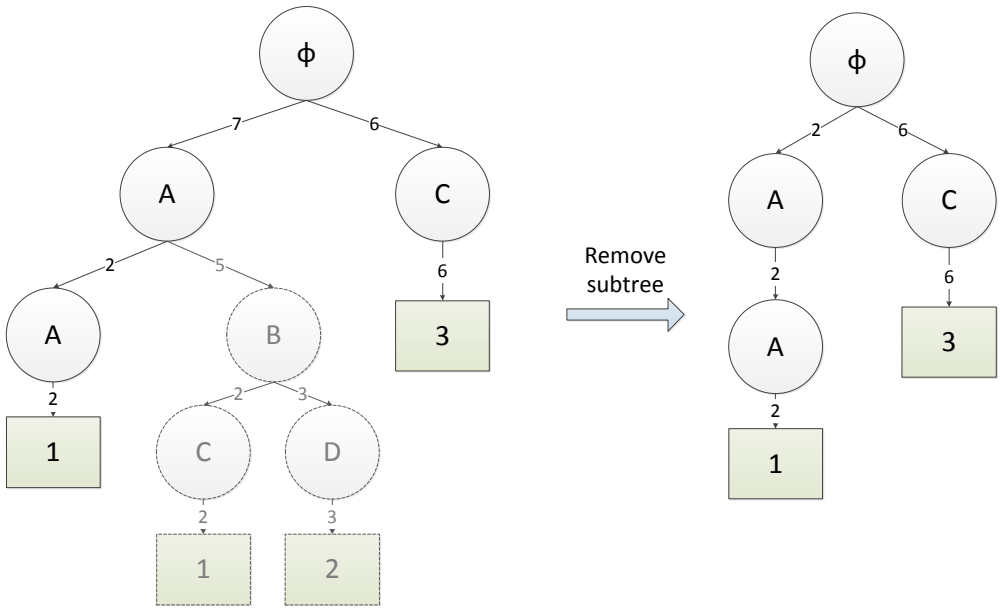

Figure 4.7: Simple example of the subtree with node $n = B$ being removed

making such distinctions and as such this is more of a preprocessing of the training data than an actual pruning method, but it can also be performed after the tree has been constructed and will result in a tree size reduction as is expected with tree pruning.

4.6.3 Size Ratio Pruning

The third method we have considered, size to frequency pruning, is a more complex pruning method that uses the structure of the sequence tree and the frequency of the different paths in the tree to influence what should be pruned in the sequence trees. Consider two sequences with unique labels; if a single sequence was removed the accuracy would be decreased by $\frac{1}{2}$. However if the removed sequence only appeared $\frac{1}{100}$ of the times in the training data, the loss of accuracy overall would only be 1%.

Using this notion a pruning algorithm could be one that iterate through a sequence tree and remove all paths that have a low frequency, but also looks at the gain of pruning a certain path. This ratio of path length to frequency will be the threshold that determines if a path is pruned or not. Consider at a node in a sequence tree we have two edges both with the same low frequency; one edge leads to a subtree that has a total of 6 nodes while the edge leading to the other subtree only consists of 2 nodes.

Clearly pruning the largest subtree will result in a smaller sequence tree but it would do so without a bigger loss in accuracy than pruning the other edge. A more efficient pruning algorithm is thereby one that can take into account the size of the subtrees it prunes.

We propose Algorithm 4 as a method to pruning sequence trees based on size to frequency pruning described previously. The method uses a threshold to determine what subtrees to prune where the threshold describes the ratio between the size of a subtree and the frequency of the edge leading to the subtree. So when specifying the

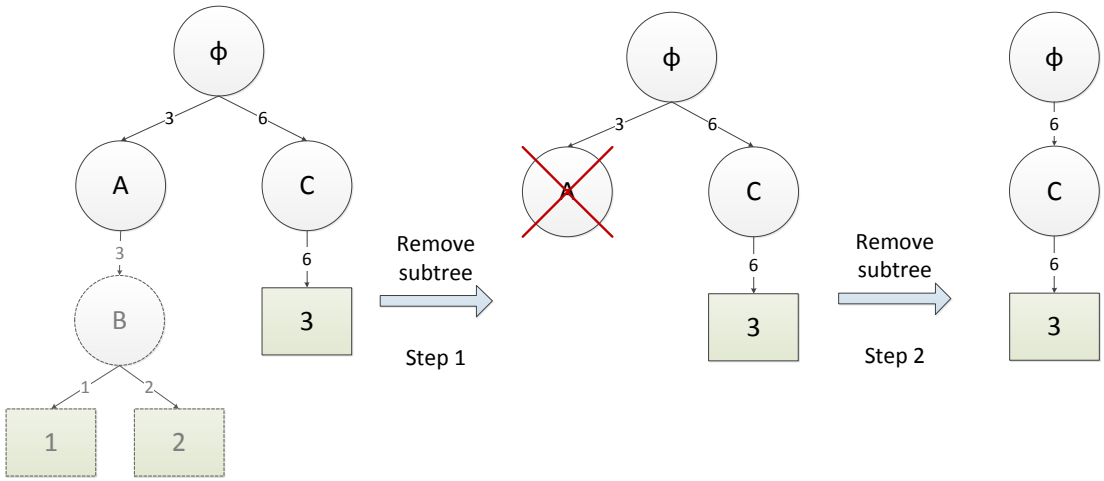

Figure 4.8: Subtree removal with its root node B being removed resulting in a whole path being removed

threshold you are in essence specifying how infrequent a path has to be in contrast to the size reduction you gain by removing it.

The algorithm performs a top-down iterative breadth-first search for any edges to subtrees with ratios at or below the given threshold and removes these subtrees. As stated this ratio is simply the frequency for such an edge divided by the number of nodes in the subtree led to by this edge. This means that that the more frequent the sequence is the longer it must be to be pruned in this approach.

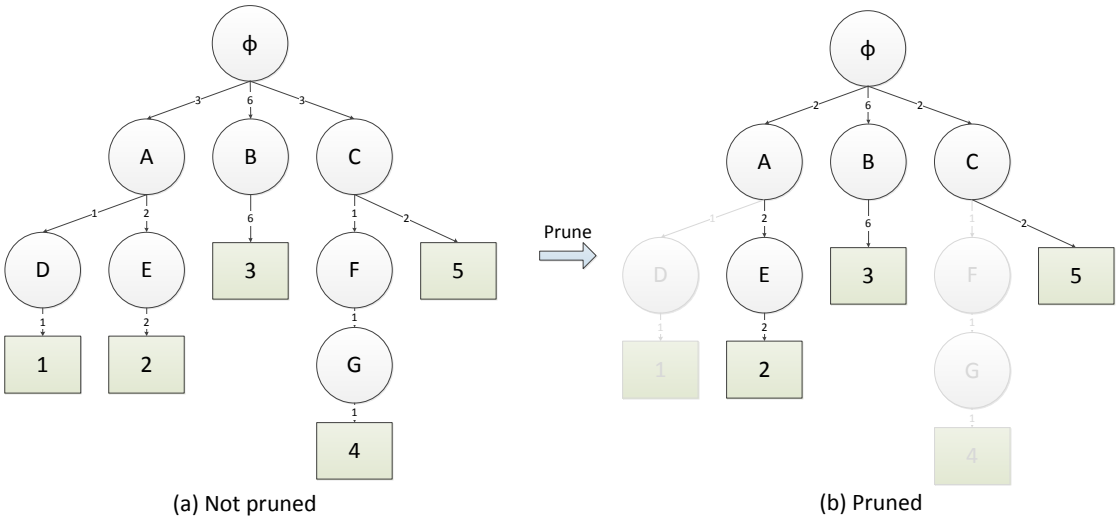

Figure 4.9: Example of a sequence tree being pruned with a threshold value of $\frac{1}{2}$

As an example of the pruning algorithm consider Figure 4.9. When we want to prune the sequence tree we must specify a threshold value. In this example we only want to prune subtrees if for each occurrence we remove at least two nodes. The threshold is defined as $threshold = \frac{frequency}{subtreeSize}$, in this example we use a threshold of $\frac{1}{2}$. Having decided on a threshold value we move on to pruning the sequence tree.

Algorithm 4: Sequence Tree Pruning Algorithm

Input: A sequence tree T, threshold value t
Output: A pruned sequence tree T

```
1  initialize openSet
2  initialize newSet
3  openSet ←(add) root ∈ T
4  while openSet != empty do
5      foreach node n ∈ openSet do
6          foreach edge e ∈ n do
7              subtreeSize ← Get subtree size of e.childNode
8              ratio ← e.frequency / subtreeSize
9              if ratio <= t then
10                 SubtreeRemovalAlgorithm(T, e.childNode)
11             else
12                 newSet ←(add) e.childNode
13     openSet ← newSet
14     empty newSet
15 return T
```

We start by going through the tree in a top-down manner starting with the root node. Each of the subtrees leading from the root node is then examined, we calculate the subtree size by counting the total number of nodes the subtree consists of, including both internal node and leaf nodes. This results in subtree sizes of 5, 2 and 5 following the edges from left to right. For each subtree we then calculate the ratio between the frequency of the subtree and the size. This gives the ratios $\frac{3}{5}$, $\frac{6}{2}$ and $\frac{3}{5}$ respectively. As none of these are below our threshold value of $\frac{1}{2}$ we do not remove any of them. Each child of the root node is examined in a similar fashion following the same procedure until the whole sequence tree is traversed and pruned. A case where the sequence tree will be pruned is when considering the subtree below node A, the subtree path to D has from this point a pruning ratio of $\frac{1}{2}$. We then remove this subtree using the subtree removal algorithm explained in section 4.5.

Using a threshold value of $\frac{1}{2}$ the sequence tree in Figure 4.9 is reduced from 13 nodes to 8 nodes, almost a 40% reduction in size. Assuming we wanted to test the accuracy of the sequence tree with the same sequences we used to construct it we would go from 100% accuracy to 83% accuracy. So in this example by pruning with a threshold value of $\frac{1}{2}$ we get a 40% reduction in size but only a 17% loss in accuracy.

Following tree pruning we proceed to describe clustering of event sequences. This is the final method we explore for reducing sequence tree size, and is essentially what was discussed in the low frequency pruning method, but is performed as preprocessing of the training data before the sequence tree is constructed.

4.7 Sequence Clustering - Preprocessed Sequence Generalization

Clustering could be used to group similar sequences in the training set together. If such a generalization can be made, representative sequences from these clusters can be used as substitutes for the original sequences when constructing the sequence tree, thus reducing the size of the tree.

Several clustering methods exist with different strengths and weaknesses, and from these we will be focusing on hierarchical agglomerative clustering, as it is quickly implemented and we can change the resolution of our generalizations.

4.7.1 Hierarchical Agglomerative Clustering

Hierarchical agglomerative clustering methods follow a basic approach described by Tan et al.[20], and the method we use shown in Algorithm 5 is no different. Each input sequence is seen as a cluster of one sequence, and these clusters are then merged together one step at a time until only a desired number of clusters remain.

Algorithm 5: Agglomerative sequence clustering

Input: Sequences S, desired number of clusters t.
Output: Clustered sequences S.

```
1 Assign a cluster to each sequence in S.
2 Compute the initial proximity matrix.
3 repeat
4 |   Merge the two closest clusters in S.
5 |   Update the proximity matrix.
6 until number of clusters n ∈ S ≤ t;
7 return clustered sequences S.
```

The use of a proximity matrix is a dynamic programming optimization to speed up the clustering process; once the distance or proximity between two clusters i and j has been calculated the result is stored in the matrix at position (i, j) for later use without needing a recalculation.

When sequences have been clustered together a representative sequence of each cluster can be found, such as by taking the medoid sequence of each cluster. These representatives are then the only sequences that will be used in constructing the now smaller tree.

The notions of closest clusters and cluster proximity have intentionally been left unexplained as these greatly affect the quality of the clustering method. In the following we first describe complete-linkage, our linkage criteria of choice for determining closest clusters, followed by describing our distance metric used to calculate sequence proximity.

4.7.2 Complete-Linkage Criteria

For the complete-linkage criteria two clusters A and B are considered closer than A and C if the maximum distance between a sequence from A and B is smaller than

the maximum distance between sequences from A and C.

Complete-link clustering thus compares the distance between the furthest sequences in two clusters, making the method prefer elliptical clusters over non-elliptical clusters and making it more robust to noisy data than other linkage criteria such as single-link.

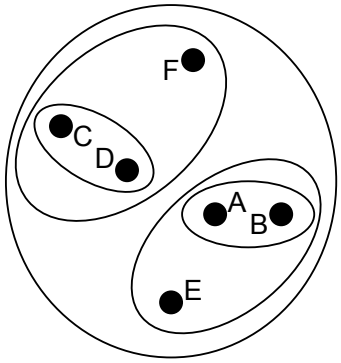

Figure 4.10: Complete-link clustering example

An example of complete-link clustering can be seen in Figure 4.10 where the distance metric is simply euclidean distance between two points in a 2D coordinate system. Clusters are grouped together one closest pair at a time, such as A and B, then C and D, and so on until all points are contained in one cluster.

4.7.3 Sequence Distance Metric

Distance between two sequences can rarely be measured in the same way as distance between points in euclidean space. Sequences only have events at different positions in the sequence and an associated label, and thus only these can be used to measure distance between sequences.

We propose the following distance metric for finding the distance between two sequences.

$$d(S_1, S_2) = \sum_{i=1}^{min(|S_1|,|S_2|)} \begin{cases} 1 & \text{if } \{s_i | s_i \in S_1\} \neq \{s_i | s_i \in S_2\} \\ 0 & \text{otherwise} \end{cases} + ||S_1| - |S_2||$$

Given two sequences S_1 and S_2 to calculate the distance we compare events at the same position in each sequence; if both events are equivalent the distance is zero, otherwise for each non-matching event the distance increases by one. We do this for every event in S_1 and S_2 summarizing the total difference. Additionally if the two sequences are of different length, the length difference is added to the distance.

The distance method we propose is a general one but many different methods can be used. One could have specific knowledge about a domain that can be incorporated into the distance function to improve clustering. It is up to the implementer to decide what method works best in their specific domain.

4.8 Summary

We have now described sequence trees, sequence tree construction and sequence insertion. We have also described various space reduction optimizations for sequence trees, including three collapsing methods, general tree pruning, and clustering of sequences.

The collapsing methods can all help reduce tree space cost without loss of prediction accuracy, and with a correct threshold the tree pruning method should be able to reduce tree size with only minor loss of accuracy. Whether sequence clustering is a useful method of space reduction remains to be seen, but all of these size reductions will be further tested in Chapter 7.

In the next chapter we look into sequence tree prediction along with tree sorting and distribution sampling, two optimizations for improving prediction speed and accuracy of our sequence tree predictor.

CHAPTER

FIVE

SEQUENCE TREE PREDICTION

As the construction of sequence trees has been concluded in the last chapter we will now describe how we use the constructed trees for sequence prediction. Furthermore we will discuss two optimizations to the prediction method. The first is sorting the sequence tree to do faster predictions, and the second is to use distribution sampling to predict nodes, to be able to choose more broadly when predicting.

5.1 Sequence Tree Prediction

When predicting, sequence trees takes a partial sequence of events as input. In the context of the previous definitions the input is the event sequence $S = \{s_1, ..., s_j\}$ of some length j. Using this input the sequence tree is traversed such that the nodes on the path in the tree match the event sequence, and a label is returned as output.

when the event sequence is not long enough to be able to reach a leaf node, that is if the length j is less than the depth of the tree, the algorithm will after tree depth j select the edge with highest frequency until a label is reached. In Algorithm 6 this prediction process can be seen.

Initially the algorithm starts at the sequence tree's root node and looks at the first event in its input event sequence. If an edge to a label exists in the current node this label is selected as the prediction result. If no label edge is found the algorithm looks for an edge to a node corresponding to the current event in the sequence. The algorithm selects the next event in the input event sequence and continues until a label is found or all events in the input sequence have been consumed. From here if we have not reached a label the most frequent child label is selected if such a label exists. This overrides the decision to walk towards the most probable label, but one could argue that labels for sequences of lengths closer to that of the input sequence are more likely predictions than frequent sequences that are much longer than the currently input sequence. If no such label exists the algorithm keeps selecting the most frequent child node until a label is found.

As a prediction example, consider the sequence tree already constructed in Figure 5.1. Given a sequence $\{A, B\}$ we want to find a label corresponding to this sequence. We start at the root of the sequence tree and check if there is a path leading to the first event in the sequence, namely A. As we can see there exist such

Algorithm 6: Sequence Tree Prediction Algorithm

Input: A sequence tree T, an event sequence s
Output: The most probable label of event sequence $s \in T$

```
currentNode ← root ∈ T
foreach event x ∈ s do
    if currentNode is a label then
        return label ∈ currentNode
    nextNode ← find node leading to x ∈ currentNode
    if nextNode does not exist then
        nextNode ← most frequent child of currentNode
    currentNode ← nextNode
currentNode ← most frequent child label of currentNode
while currentNode is not a label do
    currentNode ← most frequent child of currentNode
return label ∈ currentNode
```

a node, we follow the edge leading to this node. We then look at the next event in the sequence, namely B. Again we find that there is an edge leading to such a node, and again we follow the edge leading to that node. As all events in the event sequence have been examined and we have not yet reached a leaf node we check to see whether there is an edge leading to one. This is indeed the case as there is one leading to leaf node with the label 2. So given the sequence $\{A, B\}$ we predict the label 2.

As can be seen we are able to find an exact path through the tree, by first going from the root to the A node in Figure 5.1 step 1 and then to the B node in Figure 5.1 step 2 before finally reaching label 2. In this example the algorithm always finds a node in the sequence tree matching the current event in the input sequence and therefore never needs to go along the most frequent edges to reach a label.

A more interesting example is the case where the sequence we wish to predict is $\{B, C\}$. Here we follow the nodes until reaching node C, as seen in Figure 5.2 step 2, but contrary to the previous example we cannot read a label at this point. To resolve this we take the most frequent edge, the edge leading to node C and from here we again take the most frequent edge thereby reaching the leaf node containing the label 2.

Having described sequence prediction we now examine its time complexity to estimate how much time a prediction requires.

The time complexity for sequence tree predictions is $O(|X|l)$, where l is the depth of the sequence tree and X is the event alphabet of the sequence tree. The reason for including $|X|$ is that whenever the most frequent edge must be found the algorithm must search through all immediate edges to find the most frequent, and in the worst case this amounts to $|X|$ lookups at each level in the tree of maximum depth l.

To give a concrete example of lookup time requirements we turn to Figure 5.2 from the previous prediction example. Here we add together all the node visits

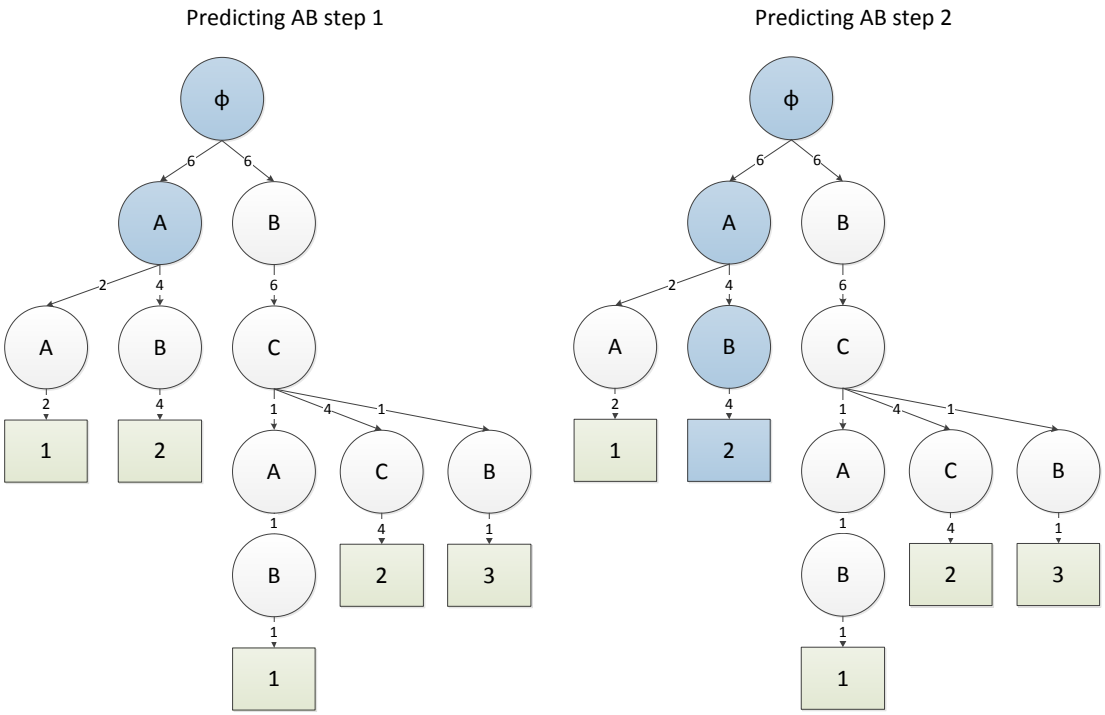

Figure 5.1: Prediction example of the event sequence $\{A, B\}$

required when searching through the sequence tree during the same prediction. We first find the node B matching the first input event, which brings visits to 1. Then we find node C which is also a match for the second input event, bringing visits to $1 + 1$. Both of these require only one visit each if event matching is implemented as a hash function allowing for direct matching of an input event to a sequence tree child node.

At this point we have no further input events to match nodes against and have to start searching for the most frequent edges; here finding the next node C requires visiting all three children to compare their frequencies before selecting the maximum, bringing visits to $1 + 1 + 3$. From here we can finally reach label 2 by another visit, bringing the total visits to $1+1+3+1 = 6$. In bigger examples the required visits of course increase, but will always be bounded by the $O(|X|l)$ time complexity based on event alphabet size and sequence tree depth.

In this chapter's remaining sections we present several techniques to improving sequence trees with regards to the prediction speed and attempt to use distribution sampling to get a more even distribution of predictions when using partial sequences.

5.2 Tree Sorting - Prediction Speed Improvement

A way to improve prediction speed is to sort the sequence tree. By sorting the tree according to a specific set of rules it is possible to reduce the time complexity in prediction time. This comes at an increased tree construction time, but prediction speed for each prediction is increased.

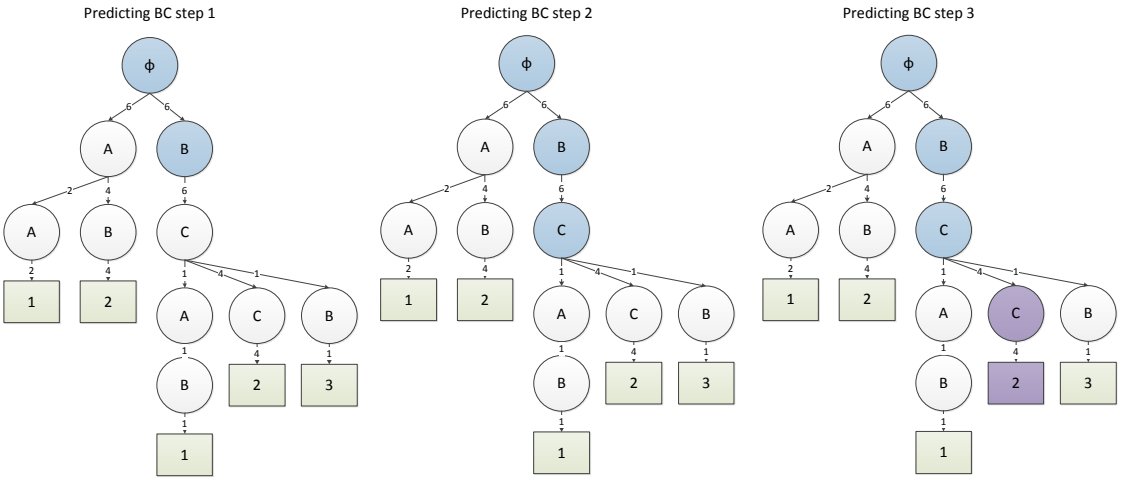

Figure 5.2: Prediction example of the event sequence $\{A, C\}$

We want to sort the tree so that the most frequent node is always in the leftmost edge. With this knowledge it is possible when making a prediction to simply take the leftmost edge without checking all child nodes to find the most frequent node.

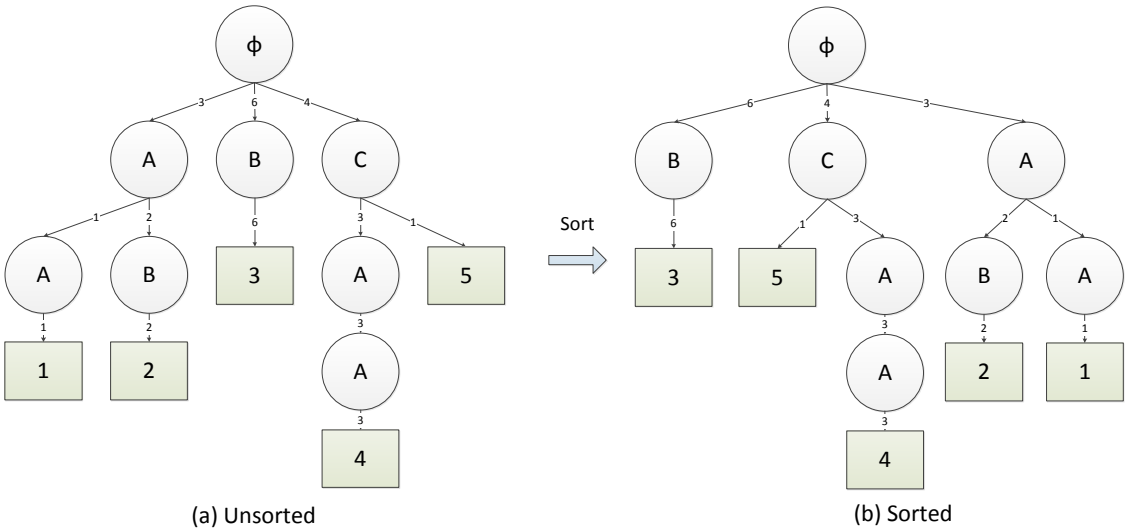

Figure 5.3: Example of a sequence tree being sorted to improve prediction speed

Consider Figure 5.3 (a) where we have an unsorted tree and we want to make the prediction with the empty sequence $S = \emptyset$. Using the normal prediction algorithm as explained earlier in Section 5.1 a prediction on this tree would result in a total of $3 + 1 = 4$ visits. The reason for needing three visits initially is that we must check the frequency of each immediate child node edge to find the maximum, but from here we only require one visit to reach the label.

Now consider Figure 5.3 (b) that is a sorted version of the same sequence tree. We can observe that each node have the most probable child node in its leftmost edge. When predicting with the same empty sequence $S = \emptyset$ again will result in a total of $1 + 1 = 2$ visits, cutting the number of visits in half.

This gives sorted sequence trees a time complexity of $O(l)$ rather than the $O(|X|l)$, thus making prediction speed independent of input event alphabet size making predictions much faster. The prediction speedup comes at the cost of longer sequence tree construction time, however. When a lot of predictions are to be made in the future the use of sorting can significantly speed up this process, but if new data will be inserted in the sequence tree on a regular basis sorting may reduce overall performance.

Another thing to note is that we sort leaf nodes with a higher priority than internal nodes. So as discussed earlier in Section 5.1 we prioritize labels where the length of the prediction sequence is the same. In Figure 5.3 this can be seen with the sequences $S_1 = \{C\}$ and $S_2 = \{C, A, A\}$.

Having discussed a method for prediction speed improvement we now look at a way to potentially increase prediction accuracy by way of distribution sampling.

5.3 Distribution Sampling - Probabilistic Selection

One possible issue with the current implementation of the Sequence Tree traversal method is that for a choice between two branches we will always choose the most likely of the two. This can mean that in principle we will never use low probability paths.

This can be a considerable drawback especially in cases where a branch contains two edges of almost the same frequency, this will in practice mean that almost half of our guesses will be wrong if we only choose the edge with highest frequency. Another case is where there is a lot of edges in a branch which is dominated by a single high frequency edge.

We will use a simple technique that in theory should in the long term give similar prediction accuracy but allows for exploration of more less frequent paths. At each branch instead of taking the most frequent path we instead calculate probabilities of the paths in the branch and use these as partitions of the interval $[0, 1]$. We sample a random number that determines which of the paths to take by which partition it falls into.

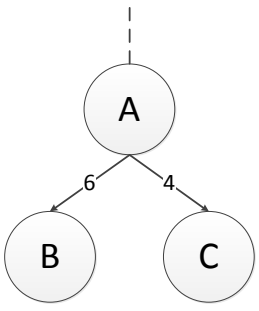

Figure 5.4: A branch with two edges

Consider a branching path with two children with frequencies 6 and 4, giving us the $\frac{6}{10}$ and $\frac{4}{10}$ probabilities of each child node respectively, as seen in Figure 5.4, we map these to $[0, 0.6]$ and $]0.6, 1]$ respectively. We sample a random number 0.3213 which falls into the interval of partition 1 corresponding to edge 1, giving us

the next step in the path. Using this sampling technique should make our guesses better reflect the probabilistic nature of the underlying sequence tree structure.

5.4 Summary

We have now described how sequence trees can be used for prediction of event sequences, along with how sorting the sequence tree can improve prediction speed and how distribution sampling may improve prediction to make more balanced predictions.

Of the two prediction optimizations the tree sorting method appears most useful; every time the most frequent edge must be followed we can reduce the visits required to just one. The distribution sampling method for selecting edges more evenly with regard to their different frequencies may improve prediction accuracy with regards to making prediction more diverse, but the method is nondeterministic in nature and as such accuracy improvements are bound to fluctuate more than may be desired.

In the next chapter we take a closer look at sequence trees and compare them to the n-gram predictor. As it turns out sequence trees are able to represent any n-gram predictor, and we will provide proof of this.

CHAPTER

SIX

MODELING N-GRAMS WITH SEQUENCE TREES

In this chapter we discuss the relations between sequence trees and n-gram predictors. We also describe how sequence trees can be used to model an n-gram predictor by way of practical examples and a mathematical proof.

6.1 N-gram and Sequence Tree Relation

N-grams and sequence trees behave in a somewhat similar way. A sequence tree could be seen as being comprised of several n-grams where the path through the tree is the sequence of the n-gram and the leaf node is the prediction. The major difference is that sequence trees combine different length histories to predict a label, and also that the label definition is much more abstract than the type of prediction used in n-grams as the prediction of n-grams must be of the same type as the events in the sequence it describes. Using n-grams requires one to define a specific length n, if there are less than n events n-grams cannot be used to predict on the sequence. Sequence trees on the other hand has no such restriction on the label type and can do prediction on any number of events up to the depth of the tree.

One can make use of open-loop n-gram prediction as described in Section 3.2.2 to predict full words. However the same method can also be applied to sequence trees by using regular sequence tree prediction on the last n elements of the sequence that must be predicted, and then recursively predict on the result.

In the sections below we will demonstrate that it is possible to convert any n-gram predictor to an equivalent sequence tree.

6.2 Converting N-grams to Sequence Trees

We postulate that The structure of an n-gram predictor can be converted into an equivalent sequence tree predictor; each n-gram can be created as a path from the root of the sequence tree. If we consider the 3-grams possible to create from the "Ich bin ein Berliner" example in Section 3.2. we can convert the n-gram models to sequences and then construct a sequence tree as per usual with Algorithm 2.

To convert an n-gram predictor to a sequence tree recall the definition of an n-gram predictor from Definition 12, where each n-gram of such a predictor is a

tuple

$$(\{s_{j-n+1}, ..., s_j\}, s_{j+1}, \omega)$$

where we have that $\{s_{j-n+1}, ..., s_j\}$ from the n-gram is the sequence with label s_{j+1} which occurs in the system with a frequency of ω. Recall that if we wanted to create a 3-gram predictor for "Ich bin ein Berliner" it would contain the following 3-grams:

tri-grams {({?,?,?},Ich,1),
({?,?,Ich},bin,1),
({?,Ich,bin},ein,1),
({Ich,bin,ein},Berliner,1),
({bin,ein,Berliner},∅,1)}

These 3-grams can then be converted into labeled sequences for use in constructing a sequence tree equivalent to the "Berliner" 3-gram predictor. The process for doing this can be seen in Algorithm 7.

Algorithm 7: N-gram to sequence tree conversion

Input: An n-gram predictor G
Output: An equivalent sequence tree T

```
1 initialize sequenceList
2 foreach n-gram g ∈ G with g = ({s_{j-n+1}, ..., s_j}, s_{j+1}, ω) do
3     for ω times do
4         sequenceList ←^{add} ({s_{j-n+1}, ..., s_j}, s_{j+1})
5 T ← SequenceTreeContruction(sequenceList)
6 return T
```

In the algorithm we create the sequence tree by converting each n-gram from the predictor G to a labeled event sequence and insert each such sequences ω times for each n-gram as input to the sequence tree construction algorithm explained in Section 4.1. For the previous "Berliner" example this gives us an n-gram equivalent sequence tree as seen in Figure 6.1.

The resulting sequence tree will have paths from root to leaf of length n. The probabilities for each leaf is the probability for the same prediction in the n-gram predictor. Additionally, in the constructed sequence tree each path may have multiple labels for one path, since multiple n-grams with the same preceding events could predict different future events.

Having given an introduction on how to convert n-grams to sequence trees we move on to give a formal proof showing that it indeed is the case that any n-gram predictor can be modeled by an equivalent sequence tree.

6.3 Formal Proof - Sequence Trees can model any N-gram Predictor

In this section we prove how sequence trees can model n-gram predictors. We determine this by examining the equivalence of these two prediction methods. First

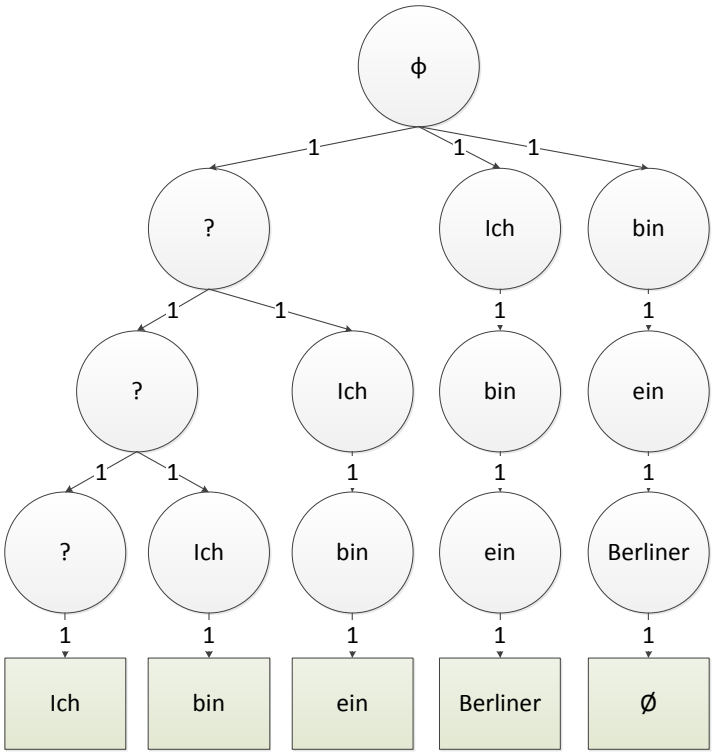

Figure 6.1: The resulting sequence tree from using the 3-grams from the example in Section 3.2

we will give a definition of how two predictors can be equivalent with regard to their input and output, and following this we show that any size n-gram can be converted to an equivalent sequence tree. Lastly we show that n-grams can also be converted to equivalent sequence trees regardless of the event alphabet size.

Together the two proofs show that we can model any n-gram predictor regardless of its degree n or event alphabet. Our notion of predictor equality is based on the equivalence of the two methods' prediction accuracy and not on structural equivalence, and we define equality between two predictors in Definition 14.

Definition 14. *Predictor Equality*
Two predictors P_1 and P_2 are said to be equal if they for the same trained input return the same output.

In this definition the two predictors are deemed equal if they on the same trained input produce the same output. Thus we focus on the output or predictions, rather than the structural equivalence of the two predictors when describing equivalence. Using this definition we proceed to prove the following theorem:

Theorem 1. *Any n-gram predictor can be modeled as a sequence tree predictor.*

In essence we want to show that sequence trees can handle the same prediction tasks as any n-gram predictor. To prove this theorem we divide the proof into two parts:

1. We show that for any $n \in \mathbb{N}$ size n-gram we can make a conversion to an equivalent sequence tree.

2. We show that the same holds for any finite E, with E being an event system.

Both proofs will be done using mathematical induction. In the first proof we induce on the size of the n-grams used by the n-gram predictor, and in the second proof we induce on the size of the event alphabet. We now begin with showing the basis and inductive step for part 1 of the proof.

Proof. Any n-gram can be modeled by a equivalent sequence tree - Part 1.
The proof that a conversion exists for each natural number n n-gram predictor to an equivalent sequence tree proceeds as follows.

Basis we show that our claim holds for $n = 1$. We disregard n-grams of size 0 as these make little sense; they would have n-gram predictors base their predicted event on absolutely no previous events, making prediction impossible.
With $n = 1$ we have the following n-gram:

$$(\{x_1\}, x_{n+1}, \omega) \tag{6.1}$$

Using Algorithm 7 we convert the n-gram to the equivalent sequence tree (N, L, E) seen in the basis of Figure 6.2, where:

- $N = \{\phi, x_1\}$
- $L = \{x_{n+1}\}$
- $E = \{(\phi \to x_1, \omega), (x_1 \to x_{n+1}, \omega)\}$

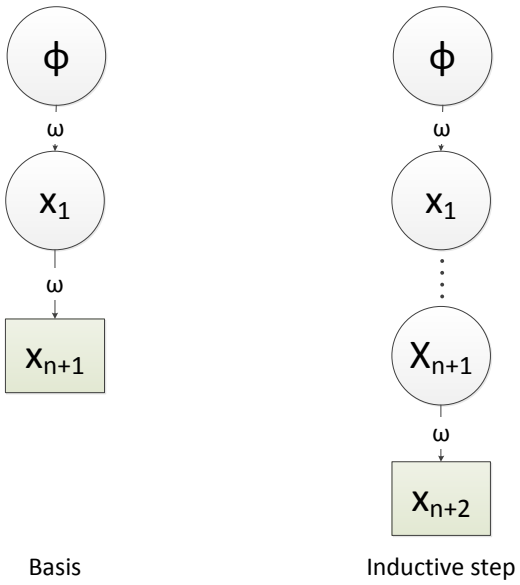

Figure 6.2: The sequence tree corresponding to the n-gram predictor of equivalence proof part 1

The two predictors are equivalent; for any input for the n-gram predictor we either find the matching n-gram or take the most frequent n-gram. In this case the n-gram

predictor will always, regardless of input give the output x_{n+1}. The sequence tree will for any input attempt to find a match for the first event, or if none is found follow the most frequent edge, which also always outputs x_{n+1}. Thus the two are equivalent as defined in Definition 14. Next we proceed to the inductive step.

Inductive step we show that if n holds, then $n+1$ also holds.
Assume n holds for some unspecified value of n. It must then be shown that $n+1$ holds, that is:

$$(\{x_1, x_2, ..., x_{n+1}\}, x_{n+2}, \omega) \tag{6.2}$$

Such an n-gram must have an equivalent sequence tree predictor. Using Algorithm 7 again we convert the n-gram to the sequence tree (N, L, E) seen in the inductive step of Figure 6.2, where:

- $N = \{\phi, x_1, x_2, ..., x_{n+1}\}$
- $L = \{x_{n+2}\}$
- $E = \{(\phi \rightarrow x_1, \omega), (x_1 \rightarrow x_2, \omega), ..., (x_{n+1} \rightarrow x_{n+2}, \omega)\}$

Using the same argumentation as in the basis step, the n-gram predictor will, if not finding a matching n-gram, select the most frequent one. In the case of sequence trees it selects matching nodes or traverses the most frequent path through the tree. Thus when we extend the length of n-grams we will still always predict the same output x_{n+2}. □

We have now shown the first half of the inductive proof of the theorem than sequence trees can model any n-gram predictor. In the following we prove the second half of the theorem concerned with the size of the event alphabet used by such predictors.

Proof. Any n-gram can be modeled by an equivalent sequence tree - Part 2.
The proof that there exists an n-gram to sequence tree conversion for any finite event alphabet E where $|E| = u$ proceeds as follows. Assume we have the following n-gram predictor.

$$\begin{gathered}(\{x_1, ..., x_j, x_{j+1}^1, ..., x_n\}, x_{n+1}, \omega) \\ \vdots \\ (\{x_1, ..., x_j, x_{j+1}^u, ..., x_n\}, x_{n+1}, \omega)\end{gathered}$$

What is most interesting here are the events x_{j+1}^1 to x_{j+1}^u; the superscript 1 to u signifies which symbol from the event alphabet E that event contains. The previous events x_1 to x_j and the succeeding events x_{j+2} to x_n may also contain any of the symbols in E, but for this proof we only concern ourselves with the different symbols for event x_{j+1}, as the same proof can be made for any of the other events in the sequence.

Given such an n-gram predictor we must show that no matter the size of $|E|$ we can convert that n-gram predictor to an equivalent sequence tree. First we prove the

basis.

Basis we show that our claim holds for $u = 1$.
With $u = 1$ we have the following n-gram:

$$(\{x_1, ..., x_j, x_{j+1}^1, ..., x_n\}, x_{n+1}, \omega) \tag{6.3}$$

where $x_1, ..., x_j, x_{j+1}^1, ..., x_n \in E$ and $|E| = 1$.
We now convert the n-gram using Algorithm 7 to the sequence tree (N, L, E) seen in the basis of Figure 6.3, comprised of the following:

- $N = \{\phi, x_1, ..., x_j, x_{j+1}^1, ..., x_n\}$
- $L = \{x_{n+1}\}$
- $E = \{(\phi \to x_1, \omega), ..., (x_j \to x_{j+1}^1, \omega), ..., (x_n \to x_{n+1}, \omega)\}$

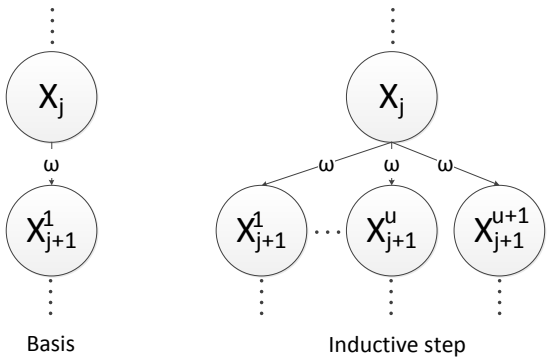

Figure 6.3: The sequence tree corresponding to the n-gram predictor of equivalence proof part 2

The two predictors are equivalent; for any input for the n-gram predictor we either find the matching n-gram or take the most frequent n-gram, similar to Proof 6.1. Again the n-gram predictor will always, regardless of input give the output x_{n+1}. The sequence tree for any input attempts to find a match for the first event, and if none is found it follows the most frequent edge instead, which again always results in the output x_{n+1}. Thus the two are again equivalent as defined in Definition 14. Now we proceed to the inductive step.

Inductive step we show that if u holds, then $u + 1$ also holds.
Assume u holds for some unspecified value of u. It must then be shown that $u + 1$ holds, given the following n-gram predictor:

$$(\{x_1, ..., x_j, x_{j+1}^1, ..., x_n\}, x_{n+1}, \omega)$$
$$\vdots$$
$$(\{x_1, ..., x_j, x_{j+1}^u, ..., x_n\}, x_{n+1}, \omega)$$
$$(\{x_1, ..., x_j, x_{j+1}^{u+1}, ..., x_n\}, x_{n+1}, \omega)$$

where $x_1, ..., x_j, x^1_{j+1}, ..., x^u_{j+1}, x^{u+1}_{j+1}, ..., x_n \in E$ and $|E| = u + 1$.
This n-gram predictor must have an equivalent sequence tree predictor. Again we convert the n-grams to the sequence tree (N, L, E) seen in the inductive step of Figure 6.3, where:

- $N = \{\phi, x_1, ..., x_j, x^1_{j+1}, ..., x^u_{j+1}, x^{u+1}_{j+1}, ..., x_n\}$
- $L = \{x_{n+1}\}$
- $E = \{(\phi \to x_1, \omega), ..., (x_j \to x^1_{j+1}, \omega), ..., (x_j \to x^u_{j+1}, \omega), (x_j \to x^{u+1}_{j+1}, \omega), ..., (x_n \to x_{n+1}, \omega)\}$

Using the same argumentation as in the basis step, the n-gram predictor will, if not finding a matching n-gram, select the most frequent n-gram's prediction as output. The sequence tree predictor again selects matching nodes or traverses the most frequent path through the tree. Thus when we extend the event alphabet size we will still always predict the same output x_{n+1} for both the n-gram predictor and its corresponding sequence tree. □

Initially we suggested sequence trees and n-grams behave in a similar way. In line with this we provided a method of constructing a sequence tree that can model an n-gram predictor, where after we provided proof of these two predictors being equivalent.

6.4 Summary

In this chapter we have shown some similarities between n-gram prediction and sequence trees. On the basis of these similarities we have attempted to prove our theorem described in Theorem 1 that states that any n-gram predictor can be modeled by an equivalent sequence tree predictor. With this interesting side note explored, the testing of our sequence prediction method sequence trees can begin in the following chapter.

CHAPTER

SEVEN

SEQUENCE PREDICTOR TESTS

In this chapter we test sequence trees as a method used for predictive text. The results are compared to similar tests done with classification trees and n-grams. We use dictionary data gathered from the British National Corpus for training and testing. We will first test the proposed sequence tree optimizations from chapter 4 and chapter 5 and reflect upon the results, and decide whether or not they will be used when we then test the spatial requirements of the predictor methods, the accuracy, and lastly the KSPC attainable by using the predictors to disambiguate words. Finally all results from the tests are summarized in a single section.

7.1 Testing Method

In this section we describe how the sequence tree, n-gram and classification tree predictors are tested. This is done via the spatial requirement, prediction accuracy and keystrokes per character tests. Prior to this, however, we carry out a range of sequence tree optimization tests to determine which parameters to use for sequence trees when comparing our predictor with n-grams and classification trees. The full range of tests is as follows:

- Sequence tree optimizations
    1. Tree collapsing test
    2. Tree pruning test
    3. Sequence clustering test
    4. Tree sorting test
    5. Distribution sampling test
- Spatial Requirement testing
- Accuracy testing
- KSPC testing
    1. Suggestion size selection

2. Keypad and touch KSPC comparison
3. Different dictionaries

Overall we perform sequence tree optimization tests to select which optimizations to use for sequence trees. Then we perform the spatial requirement, accuracy and KSPC tests. The procedure and reason for each of these tests is explained below.

The Optimization Tests

First we test the five different sequence tree optimization methods. This includes testing of the three methods for tree collapsing, tree pruning and sequence clustering to reduce the spatial requirements of sequence trees, as well as testing of tree sorting to improve prediction speed, and testing of distribution sampling with regards to prediction accuracy. In the end we decide on which of these optimizations to use to optimize sequence trees for the remaining tests, based on the performance results given here.

We perform these tests to ensure we only use those optimizations that benefit sequence trees in the remaining tests when comparing our method with classification trees and n-grams. If we do not do this we risk including optimizations that in the end can prove detrimental to sequence trees performance in the test domain of predictive text.

The Spatial Requirement Test

For spatial requirement testing we examine how dictionary size affects the spatial requirements of each predictor, focusing mainly on how larger dictionaries cause the different predictors' underlying models to take up more space for training data and information storage.

This is done to know how well the different predictors' spatial requirements scale depending on dictionary size. The lower the spatial requirements are the more sequences can be included in the model and thereby overall prediction accuracy is increased.

The Accuracy Test

Accuracy testing is where we compare the prediction accuracies of the three prediction methods. We here give an overall evaluation of how each predictor performs in different situations depending on how much information is given to the predictor. In the worst case predictors are only given the first 10% of a word to base their predictions on, and from here we go by 10% increments until predictors are given all 100% of the word as input, in which case all predictors ideally should be 100% accurate. An example of this 10 increment predictor input could be the word "house"; the first two tests with 10% and 20% of the word would only reveal "h" to the predictor, and the next two for 30% and 40% would reveal "ho", then "hou" for 50% and 60%, "hous" for 70% and 80%, and finally "house" at 90% and 100%.

This accuracy testing with increasing amounts of input information gives us a good idea of how well the different predictors perform, both in the early steps of

prediction where little to no information on the current intended word is available, and in the late steps when nearly all of the intended word is known in advance.

The KSPC test

KSPC testing is where we finally evaluate how sequence trees perform compared to n-grams and classification trees when used to suggest a list of words to the user for predictive text messaging on mobile phones. This also includes how different list navigation approaches can affect the KSPC rating, but the main focus is determining how well sequence trees perform as a predictive text method.

KSPC testing is done because it provides information on how well each predictor would perform in an actual usage scenario when the user is typing words into the predictive text system.

Further details for each of these tests will be given as the tests are encountered in this chapter. Having described the overall method of testing we next describe the material we use as training data for the predictors during testing.

7.2 Test Material

All tests will be performed on dictionary data from the British National Corpus[15]. The BNC originally contains 938971 different words with assigned frequencies derived from a combination of context governed meeting transcripts, regular demographic mobile text message conversations and various popular written texts, and as such should cover most uses for predictive text. The different corpus are given the following names within the test chapter:

- "all" contains all of the corpus words from the below corpus subsets
- "context" contains words taken from various subject governed meetings
- "demographic" contains words from day-to-day mobile text message conversations
- "written" contains words from various popular written literature

Many of the words in either of the BNC are acronyms and there even exist onomatopoeia, words describing various sounds. Each word has an assigned frequency where the most frequent is the word "that" with a frequency of 6187267. There are however also a lot of rarely used words with frequencies at or near 1. We have chosen to remove words with symbols and to force all letters to be lowercase to simplify the tests.

We also ensure that all words are unique in the dictionaries. Thus a corpus of, say, 100 words will hold 100 unique words, but with each unique word having an assigned frequency the dictionary is actually larger if we also look at these repetitions caused by word frequencies. Whenever we in the test chapter mention dictionary size, we however always refer to only the number of unique words in that dictionary.

Finally we sort the corpus by word frequency to ensure that the most frequently used words are listed first. For each of the coming tests we can then select the n

most frequent words from this corpus to use as training data. For testing, unless stated otherwise, we will always use the "all" corpus. For the KSPC tests however, in addition to testing with the "all" corpus we also test using the context governed, demographic or written text subsets of the complete BNC to determine if different word contexts affect the predictions.

In addition to choosing what context to derive our training data sequences from, we must also decide how we wish to encode these sequences. In the case of predictive text there are two different input types, alphabetic or "numberfied" text input, as described in Section 1.3. Alphabetic input is straight forward to predict on, but numeric input however adds an additional layer of indirection due to ambiguity. With numeric input we must not only predict the next characters, but we must also predict which character the user is inputting at each step when he or she presses a number button on a mobile phone.

By way of a small test in this section we will now explore the loss in accuracy by using numeric input as opposed to regular text input. The test has been performed on a dictionary of 1024 regular or "numberfied" words, where we use the dictionary as training and test set. We try to predict on only the full words in the dictionary, so no predictions are made on partial sequences. In Figure 7.1 the effect of alphabetic and numeric input on prediction accuracy can be seen.

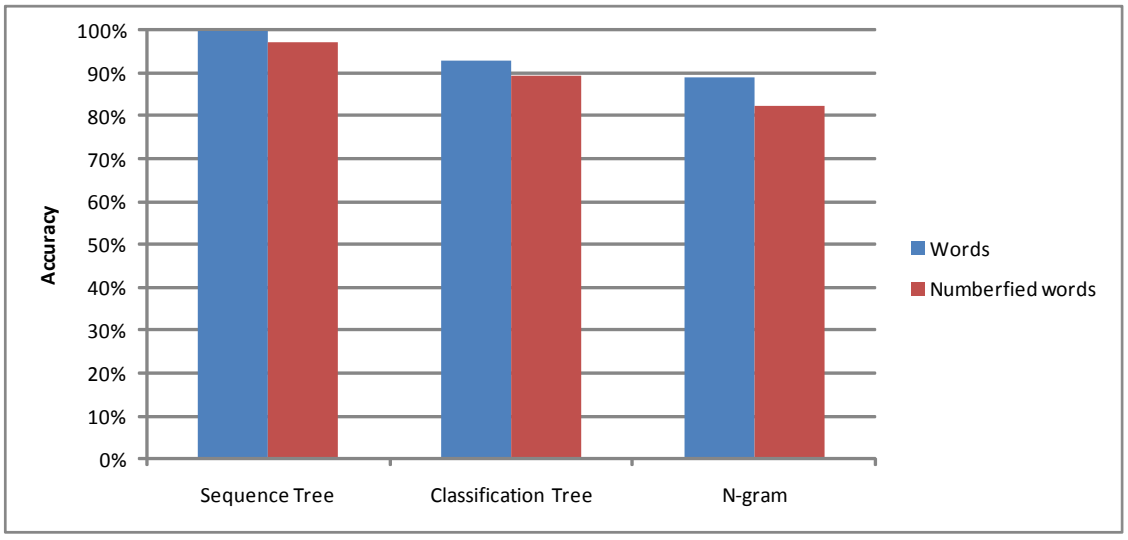

Figure 7.1: Classifier accuracy on regular and numberfied dictionary

We measure prediction accuracy by how many of all predictions on the training set are correct predictions. Thus if 64566 predictions are made in a prediction test run and all 64566 are correct we will get an $accuracy = 100\%$. One should note that the number of predictions to perform is not the same as the number of words in the training data; each word in the corpus has an assigned frequency, and prediction on that word is repeated as many times as its frequency. This is done to make more frequent words' predictions mean more to predictor accuracy than predictions on words that only appear rarely in the corpus. This prediction accuracy can be seen in Definition 15.

Definition 15. *Accuracy Quality Metric*

$$Accuracy = \frac{\sum_{w \in TestData} w_{frequency} \cdot \begin{cases} 1 & \textit{if prediction is correct} \\ 0 & \textit{otherwise} \end{cases}}{\sum_{w \in TestData} w_{frequency}}$$

As per the definition we sum over each word in the test data, multiplying the word's frequency with 1 if its prediction was correct or 0 otherwise. This sum is then divided by the summed frequency of all words in the test data, resulting in a ratio of correct predictions over total predictions. Having defined this prediction accuracy we now return to testing how numeric input affects prediction accuracy.

In all cases accuracy drops by between 3% and 7% for numeric input depending on the prediction method used. As can be seen in Figure 7.1 sequence trees have 100% accuracy while using regular words, however on numberfied words the method has a small decrease in accuracy. The drop of accuracy in sequence trees is caused by words that have the same numberfied description; these words will have two or more leaves for the exact same path in the tree, and witout more information the two cannot be distinguished from each other. Therefore at some point sequence trees produces a wrong prediction when it has to choose one of the two without additional information available.

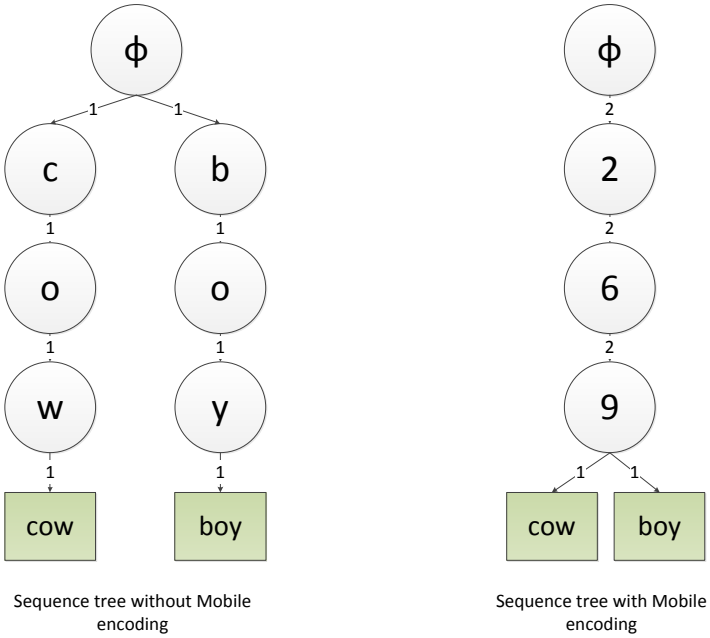

Figure 7.2: Sequence tree text reduction to mobile encoding

Consider as an example the sequence tree in Figure 7.2. In the first sequence tree there are two distinct paths leading to both "*cow*" and "*boy*". However when the sequence tree is constructed using numberfied words, as is the case with the second sequence tree in Figure 7.2, both labels now have the exact same path. This results in an ambiguity that cannot be distinguished between without further information, resulting in a lowered accuracy.

Regardless of the drop in accuracy, numeric input also most closely resembles how input is given to predictive text systems on mobile phones. To have a test that

is close to how text input is on mobile devices we have chosen to use numeric input in our tests. As described in this section using numeric input decreases the accuracy of our predictions, but is a much more realistic test case when we use this to do the accuracy and KSPC testing of the three prediction methods.

7.3 Test Implementations

In this section we briefly describe the implementation details of each of the three predictors used in testing. This includes important details about sequence encoding as well as parameters affecting sequence prediction for each of the three methods.

7.3.1 Classification Trees

The classification tree predictor uses the J48 classifier in the WEKA toolkit which is an open source implementation of the C4.5 classifier described in Chapter 3. Sequences are encoded as a range of individual events, with one event for each position in the longest sequence from the training data. If a sequence given as input during tree construction is shorter than the longest sequence all of its non-existing events are represented by a special blank event. Later, during prediction, currently unobserved events for a sequence to predict are instead encoded as the special "?" Weka symbol rather than blank events. This is done since the predictor cannot know for certain if the remaining events for the input word are letters or non-existing in the case of words shorter than the longest word.

We use the default values of classification trees given by the J48 algorithm in Weka. This includes a minimum number of two objects per leaf, in effect removing all sequence that occur only once in the training data, as well as setting a confidence factor of 0.25 for tree pruning. This confidence factor acts as a threshold where smaller values will cause more aggressive pruning.

7.3.2 N-gram Predictor

The n-gram predictor has been implemented by using 3-grams of the previous letters in a sequence as basis for prediction.

For our n-gram predictor we have chosen to use 3-grams, where we use three events to make a prediction. Using only 2-grams we would only be able to store 676 unique combinations. With very large dictionaries that we wish to test on 65.000+ words this would probably not be enough to distinguish words efficiently, therefore we have chosen to use 3-grams where we have 17576 distinct combinations. Additionally we have opted not to use hierarchical n-grams as we use our special definition of using "?" symbols when we otherwise would not have enough events to construct a 3-gram i.e. the 3 first events of each sequence.

We use the open loop prediction method described in Section 3.2.2 which allows us to predict whole words rather than just the next character. The three last events in the sequences given as input are used to find a suitable n-gram to start the prediction process, and then for each step the most probable events are selected and used to make new predictions. The prediction process stops when a whitespace is

predicted, as we use this as a terminal symbol for full words. The n-gram predictor additionally halts after either finding a number of suggestions or running for a number of cycles to avoid infinite loops of prediction, and then outputs the most probable complete words found. If the input sequence was numberfied the output is then run through a dictionary to look for a word label matching the predicted character sequence. This is needed for converting numberfied predictions into real words.

7.3.3 Sequence Trees

The sequence tree predictor has been implemented according to the specifications in Chapter 4 and Chapter 5. In our implementation each optimization method can be applied separately or in any configuration to a sequence tree such that we can measure the performance of them individually. So as an example we could choose to sort and collapse paths in one configuration and in another use only sorting.

Only the optimizations selected during optimization testing in the next part of this chapter will then be enabled for the predictor during the spatial requirement, prediction accuracy and KSPC tests. In the following section we will test each of these optimizations and determine which of them will be used for the remainder of this test chapter.

7.4 Sequence Tree Optimizations

Following the test material and predictor implementation details we now proceed to test the optimizations presented for sequence trees. We will test the following optimizations:

- Collapsing methods
- Tree Pruning
- Clustering of sequences
- Sorting
- Distribution sampling

We will for each optimization conclude on their effect on predictor performance and decide on which of the techniques to use in further testing depending on the results.

7.4.1 Tree Collapsing Test

The first optimization test we will do is testing whether or not it is possible to gain a reduction in size of sequence trees by collapsing paths, subtrees and combining labels as described in Section 4.4. We expect that as the domain contain many distinct sequences with little branching there may be many opportunities to collapse paths in sequence trees. Collapsing subtrees and combining labels will not give much in

the domain, as all labels are unique. To calculate the reduction in size we define the prediction space usage Definition 16.

Definition 16. *Size definition*
Sequence tree size: S_s = *number of nodes in the tree*

Size is measured by the number of nodes and used by the sequence tree to store its underlying learned data structure.

The different collapsing methods have been tested on a dictionary size of 65536 words as both training and testing data. The results can be seen in Figure 7.3, and the reduced sizes can be seen in Table 7.1. To recap the collapsing methods:

- **Collapsed Paths**
  Collapsing paths that do not branch; i.e. when we are sure which node we will be in after a number of steps.

- **Collapsing Subtrees**
  Collapsing subtrees that lead to the same label; i.e. when all predictions inside that subtree return the same label.

- **Collapsing Labels**
  collapsing identical labels, such that multiple sequences refer to the same instance of that label.

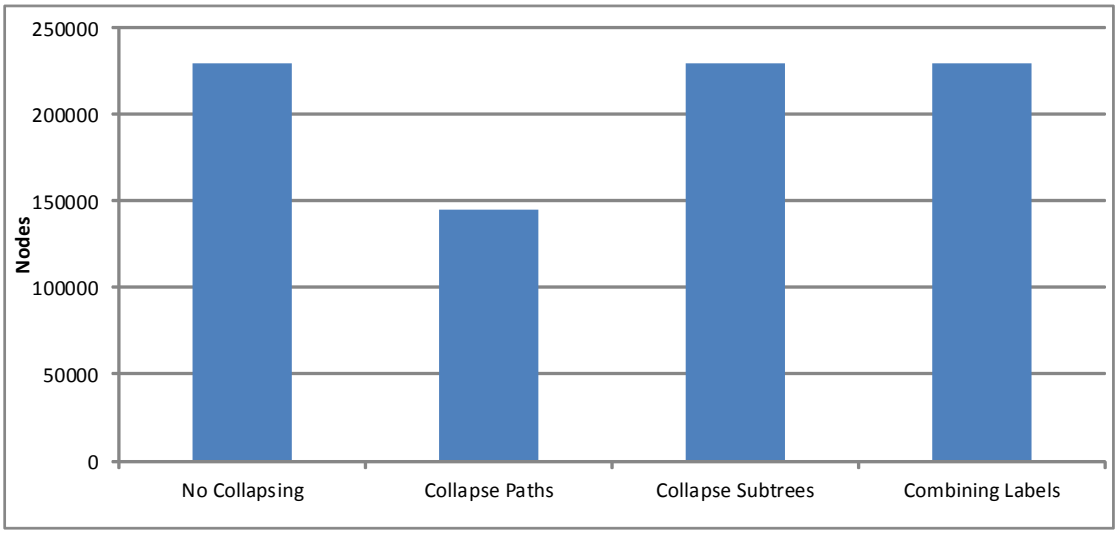

Figure 7.3: Sequence tree size using different collapsing methods

Of the different collapsing methods the only one to reduce the tree size in this predictive text domain is collapsing non-branching paths. We manage a reduction to 63.4% of the original size of the tree which is a significant improvement. As can be seen in Table 7.1, collapsing subtrees, and combining labels does not give any reduction.

	Uncollapsed	Collapsed paths	Collapsing subtrees	Combining labels
Size	100%	63.4%	100%	100%

Table 7.1: Sequence tree collapsing methods showing reduction as size of the original

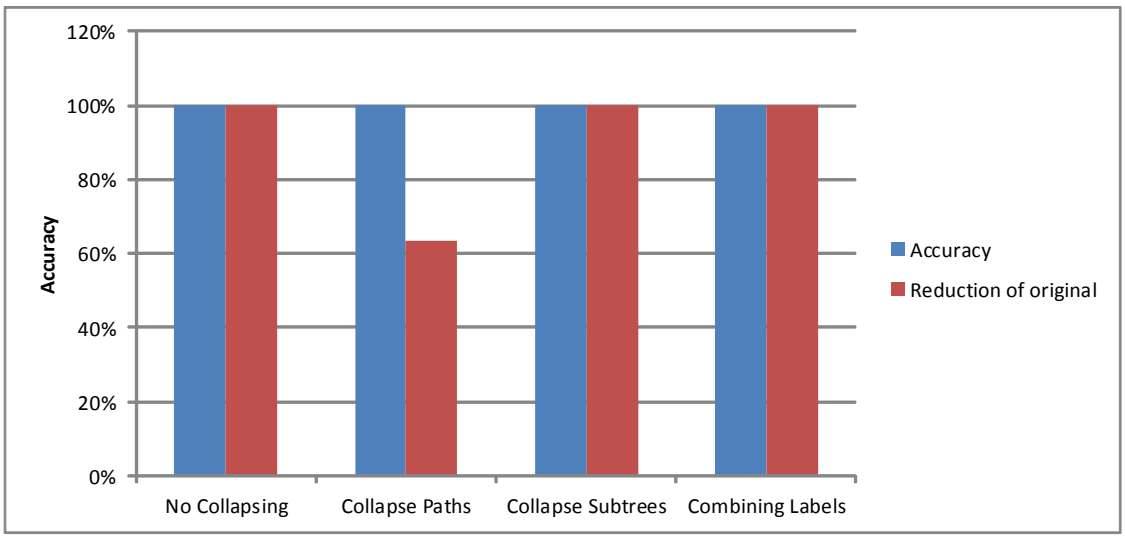

Figure 7.4: The effect of collapsing on prediction accuracy.

In our test data there cannot be two identical labels, as no two words are alike, therefore in this particular domain these two collapsing methods cannot give a reduction in the tree size. In other domains where labels are more general these reductions could possibly provide a significant reduction in tree size.

One situation where the other collapsing methods can aid in tree size reduction is when common typos are included in the dictionary as well, for instance when two words "wich" and "which" leading to the same label "which" are both in the dictionary. Adding these typos will increase the dictionary size but also increase the power of the predictive text method to let it be able to do some minor spelling correction while predicting. In these cases performing path and subtree collapsing the size increase can be minimized, as it may be possible to collapse typos and words into smaller or fewer subtrees.

As previously argued collapsing does not affect the prediction accuracy, therefore we have sampled the accuracy for collapsed sequences trees, the results can be seen in Figure 7.4. The accuracy is at 100% for all cases of collapsing, which is the same as for the uncollapsed sequence tree. We will use all the collapsing methods in the remaining tests as they are quite lightweight to use and they reduce the size of the sequence tree without changing the prediction accuracy. In the next section we will test another method to reduce the tree size, this time however we will modify the prediction accuracy as we will attempt to prune low frequency subtrees from sequence trees and see what we can gain in size at the cost of prediction accuracy.

7.4.2 Tree Pruning Test

In this optimization test we will use the pruning technique from section 4.6. We will measure both the reduction in size and how it affects the prediction accuracy, as the optimization is not lossless in its precision. We expect pruning to drastically reduce the size, however we cannot predict how it will reduce the accuracy of sequence trees. We use the size ratio pruning technique where we prune a sequence if the ratio between its frequency and its size is blow a certain threshold.

We have tested pruning on a dictionary size of 65536, used both as training and testing set. We tested with a range of different threshold values to see the effect on both accuracy and size. The results can be seen on Figure 7.5.

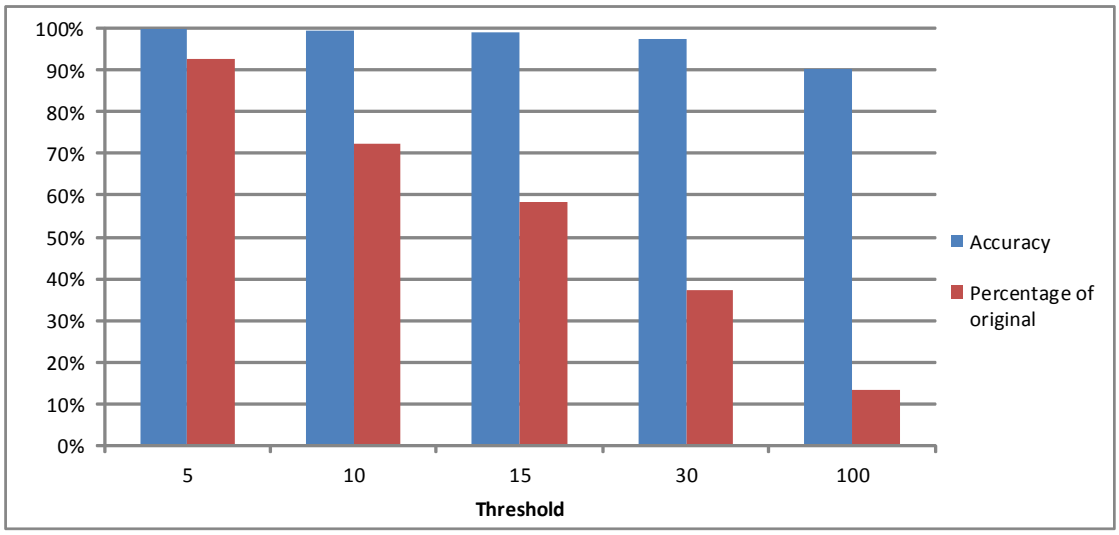

Figure 7.5: Sequence Tree Pruning with accuracy and reduced size as threshold value increases

We can see that as the pruning threshold value is increased the size of the sequence tree drops significantly without a large loss in accuracy. The very low reduction in accuracy may be because of the dictionary data; short words like "as", "is", and "in" all have high frequencies whereas longer words often have very low comparable frequency. The high frequency words keep the accuracy up while the low frequency words can be removed without much impact overall in the prediction accuracy.

Consider the threshold value of 100. Recall from Section 4.6 this means that we prune the tree every time an edge has a frequency to node ratio of 100:1. The high ratio is due to the training data; as mentioned in Section 7.2 words may have a frequency of over 6 million, and therefore a large threshold value is needed as the dictionary contains entries with high frequencies for some words.

As can be seen a threshold value of 100 gives a reduction in accuracy of about 10%, however the tree size is reduced to a mere 13% of the original size. This is possible in part due to the training data where a relatively small part of the data represents the majority of occurrences, and this allows the pruning algorithm to thin out the largest subtrees that do not affect the overall accuracy by a large degree. Using the same pruning on other data sets with a more even distribution

Clusters	cluster 1	cluster 2
Words	Hello	Big
	Yellow	Jig
	Mellow	Pig
Representative	Yellow	Jig

Table 7.2: An example of two clusters of words

of frequencies would surely not give this impressive results, however we see that when this is not the case the pruning method proposed for sequence trees is highly effective.

We will not use pruning in the remaining tests even though it shows very promising results. The reason for this is that pruning only produces an approximation of the original sequence tree, this conflicts with the wish to compare the predictors on the same training data and the same potential predictions. With pruning we will not be able to explore the best possible accuracy achievable with sequence trees. In the following section we will look at another lossy optimization to reduce the tree size using clustering of sequences.

7.4.3 Sequence Clustering Test

In this final space optimization test we will experiment with using clustering to reduce the size of sequence trees. If a suitable distance function and a way of computing an average representative for each cluster could be found, then we expect this optimization to be feasible.

However in predictive text clustering is not a good solution to reduce the tree size. In the case of the dictionaries and word data we use in the tests clustering will greatly reduce the accuracy of the sequence trees as we cluster the training data.

The problem is to create a candidate for each cluster; for words there obviously does not exist an average word representative. To recap, if we used the distance metric from Section 4.7, we calculate the distance between two sequences as the number of events they differ by at each position in the sequence plus the difference in length of the sequences. To exemplify how clustering does not work well using text data consider the two clusters described in Table 7.2.

As the table shows if we choose a representative from each cluster we will get an error of $\frac{2}{3}$ for each cluster. This optimization results in roughly the same as reducing the dictionary size to the number of clusters found, if a representative is picked from each group.

We have tested the clustering optimization on a dictionary with 1024 words. As anticipated the results, seen in Figure 7.6, are not very good. For a good compression method we would like to have the size fall more than the accuracy when we cluster, as is the case when we use pruning above. However clustering gives a very large amount of error for small reductions in size. With 200 clusters the size of the sequence tree is reduced to 37% but the accuracy is reduced to 10%, which is highly unacceptable.

Clustering can possible be used where the data has more similarities and each sequence does not have a unique label. This is not the case in predictive text and because of the poor results in our domain we will not use clustering for the remaining

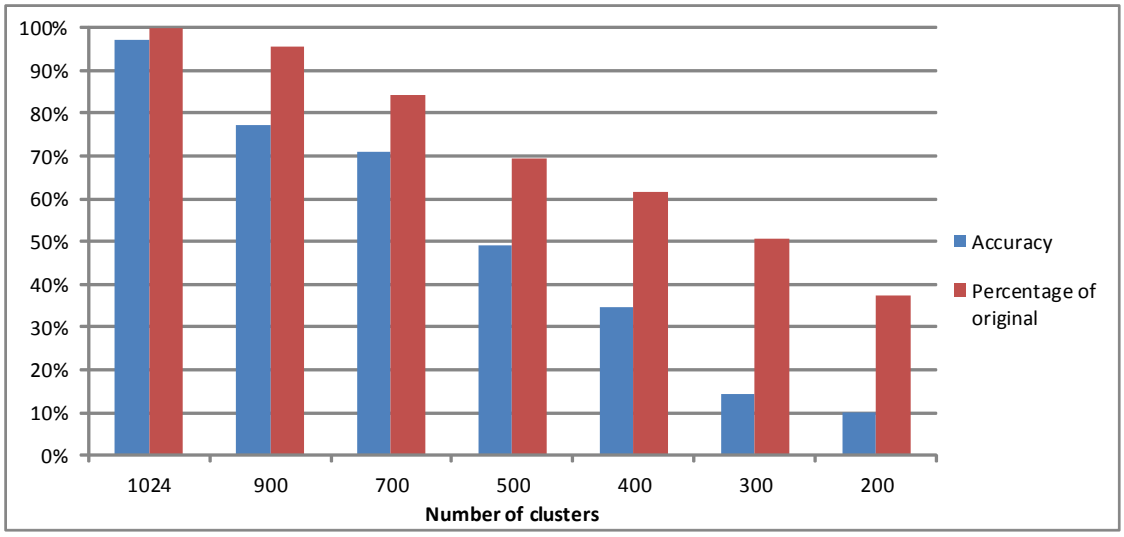

Figure 7.6: The effect of clustering training data on both size and accuracy of sequence tree models

tests. The next optimization we will discuss is how to increase the speed of sequence trees using sorting.

7.4.4 Tree Sorting Test

In this optimization test we will use sorting to increase the prediction speed. Without this optimization we need to, when we have to predict a new node, go through all child nodes to find the most frequent child. If we sort these such that the leftmost node in the tree is the most frequent, we can reduce the number of visits per node prediction greatly, as we can instead just choose the left most node in the tree. We have tested sorting on a dictionary size of 16384 words, used both as training and testing set.

We measure the time used to predict as the number of nodes that has to be visited to make a prediction. Visits increase by one every time the sequence tree classifier traverses a node in its tree. The number of visits summarized to a total for all predictions on the test set and an average is calculated.

Recall that the time complexity for sequence tree predictions was $O(|X|l)$, where $|X|$ is the event alphabet size and l is the maximum length of a sequence.

By sorting the tree we reduce the time complexity to $O(l)$ as it is no longer necessary to check each edge of a node at each depth level of the tree. There is a catch, however, as the sorting algorithm is very time consuming. Therefore we have limited dictionary size to 16384 words for easier and more rapid testing.

To see any improvement for this optimization the sequences that is used to predict on must be of shorter length than the tree depth. The shorter the sequence is the higher reward sorting gives us in terms of reduced amount of visits, compared to linearly checking all children of a node. For this reason we have used the approach described in Section 7.1 where we divide words into ten partitions of 10%, 20%, 30%, ..., 100% of the word length, which is used to predict on.

	Unsorted, Uncollapsed	Sorted	Collapsed	Sorted, Collapsed
Visits required	100%	51%	88%	43%

Table 7.3: Percentage of visits required for prediction after tree sorting or collapsing

After all predictions are done and the number of visits counted, the average number of visits for the whole test can be calculated. In Figure 7.7 the amount of node visits needed for predicting all words in a dictionary of 16384 words can be seen, and in Table 7.3 the percentage of visits required after sorting and collapsing can be seen. The two sorting tests with collapsing have been added to see whether there is a synergy effect from using both optimizations at once.

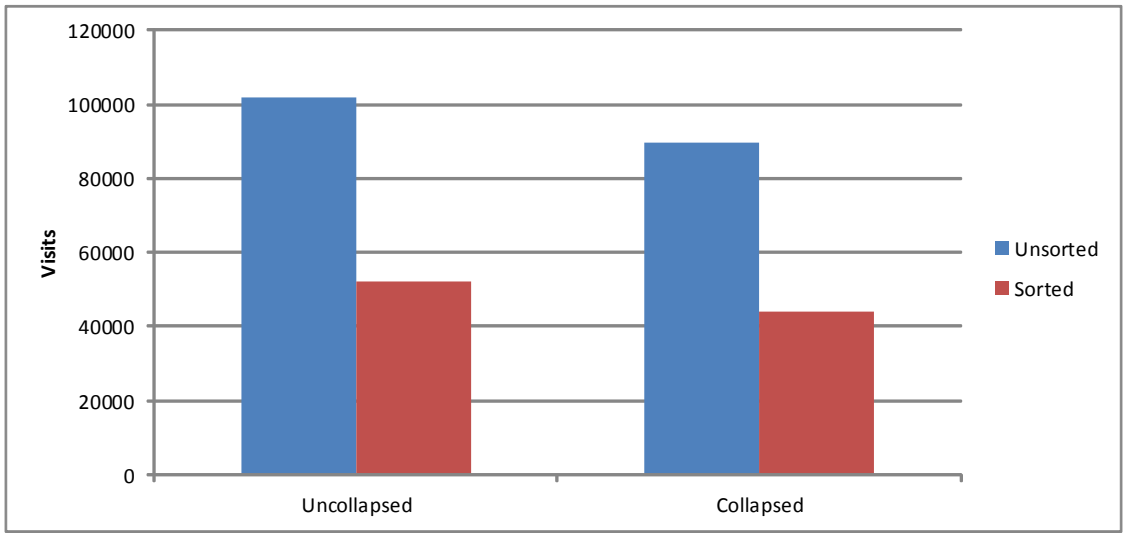

Figure 7.7: Sequence tree node visits with and without sorting

On uncollapsed trees sorting gave a reduction to 51% of the original number of visits. For the same sequence trees where the collapsing optimization has been used we manage to reduce the number of visits to 49% of the collapsed tree's original number of visits. The reduction in the number of visits by sorting is quite impressive. As we remove the need to check through each child to find the most frequent one we reduce the number of visits drastically, as Figure 7.7 clearly shows. We will not use sorting in remaining tests, as sorting does not have any effect on the spatial requirements, accuracy or KSPC. However one should use sorting when prediction speed is of importance.

The next optimization we will test is to use distributed sampling for altering the prediction accuracy of sequence tree.

7.4.5 Distribution Sampling Test

In this final optimization test we will use distribution sampling to have the predictions of our sequence trees better reflect the underlying distributions, as described in Section 5.3. The distribution sampling is used when we reach a point where we have to choose which way to go in the sequence tree.

When this happens we randomly sample one of the unknown children of the current node based on their probability. With this optimization the prediction method will sometimes choose another node than the most frequent child node contrary to the usual sequence tree prediction method. The test is done on a dictionary size of 64566 words. In Figure 7.8 the results of the test can be seen.

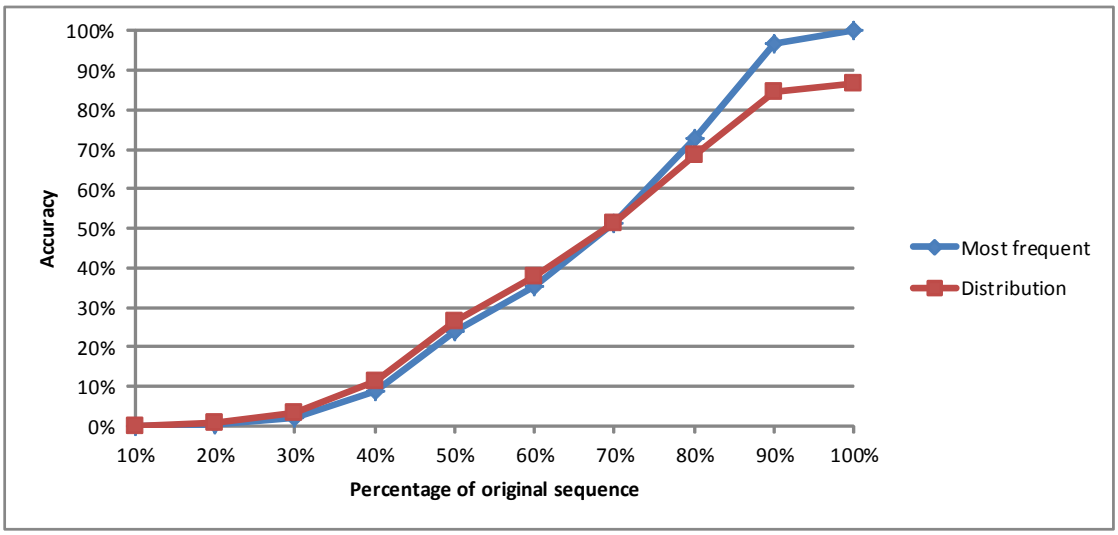

Figure 7.8: Sequence tree accuracy with most frequent or distribution selection with increasing input information

For most frequent path selection we recorded the accuracy of the predictor when always selecting the most probable path during prediction. For the distribution selection we recorded the average accuracy of the predictor over 10 prediction runs with the predictor selecting the next path according to distribution sampling. We have several tests and calculate an average accuracy as the selection process is random and may in some cases give some very bad results or very good results, and we wish to know how it does in the average case. In both cases we tested prediction with ten partitions at increasing percentages of 10% to 100% of the word size given as input to the predictor.

When all letters are known in advance the most frequent approach reaches a prediction accuracy of 100% whereas the distribution method does not. As can be seen, however, distribution sampling can indeed improve prediction accuracy when fewer letters are known in advance. Still, in this test distribution sampling only improved accuracy by a few percent compared to simply always choosing the most probable path.

Thus distribution sampling seems to mainly benefit prediction when few of the word's letters are known in advance. Finally, if distribution sampling is used when producing a list of predicted words to the user, the list may change every time a new prediction is made due to the random sampling involved.

Early on accuracy is improved by distribution sampling, but if this list of words changes constantly during text input the user may become frustrated with a word having different predictions at different times. As the results were not that impressive, and having nondeterministic predictions may have some issues. Thus we have

chosen not to use this optimization in the remaining tests.

7.4.6 Optimization Selection Summary

We have tested all optimizations, and the collapsing methods are the only one that will be used throughout the remaining tests as they reduces tree size without any loss in accuracy. We have found that tree pruning can drastically reduce the size of sequence trees at a very low reduction in accuracy. However we have chosen not to use it in the tests where we compare sequence trees to classification trees and n-grams because we want to explore the best accuracy possible using sequence trees.

As sorting does not effect spatial requirements, accuracy or KSPC we have chosen not to use it in further test, however if a sequence tree was to be used for real purposes rather than testing sorting should be used, as it results in a very large speedup during prediction.

Clustering gave very poor results, reducing the accuracy of sequence trees with only a minor spatial reduction. Lastly distribution sampling did not provide any increase in accuracy and the nondeterministic outputs could be an issue.

As stated we will only use collapsing methods for the remaining spatial requirement, accuracy and KSPC tests. With these choices made we can begin testing to see how our optimized sequence trees fare against classification trees and n-gram predictors in terms of spatial requirements.

7.5 Spatial Requirement Testing

Now that all parameters are set for sequence trees we can begin to examine the spatial requirements of the method. In this section we will test and measure how much space each prediction method uses as the dictionary size grows. The spatial requirements and growth for each method is key to having a good predictor; the less space the method require allows it to be used in more cases. Also the more we can compress data we can increase general predictive power of our three prediction methods as larger dictionary sizes can be used as training data. The larger the dictionary the more different words the predictors can recognize and predict.

It is therefore interesting to find the growth rate to see how large a dictionary we can use as training data before the methods require too much memory to be feasible.

We measure the size of sequence trees as described in Definition 16. To recap the size of a sequence tree is the number of nodes in the tree. The sizes of classification trees and n-gram predictors are measured as described in definition 17 and 18 respectively.

Definition 17. *Classification Tree Size*
Classification tree size: S_d = number of nodes in the tree

Definition 18. *N-gram Predictor Size*
N-gram predictor size: S_n = number of n-grams multiplied by $n + 1$

The first test on the spatial requirements of the three predictors has been done on dictionaries of sizes up to 1024 words and can be seen in Figure 7.9. The limit

of 1024 words is imposed by classification trees as for dictionary sizes above 1024 it is simply intractable to train classification trees using Weka and the J48 algorithm on our training data. We suspect that the very large number of information gain calculations is part of the problem as well as the method continuing to construct a tree until everything is completely classified, which obviously results in very large trees, before pruning. Both of these takes up huge amounts of memory when training on the dictionary data. The very low size restriction of classification trees severely limits its usefulness, as most dictionaries contain vastly more words than 1024. We will continue to compare sequence trees with classification trees in the following tests even though we do not believe it is a realistic approach to doing predictive text based on the results in this test.

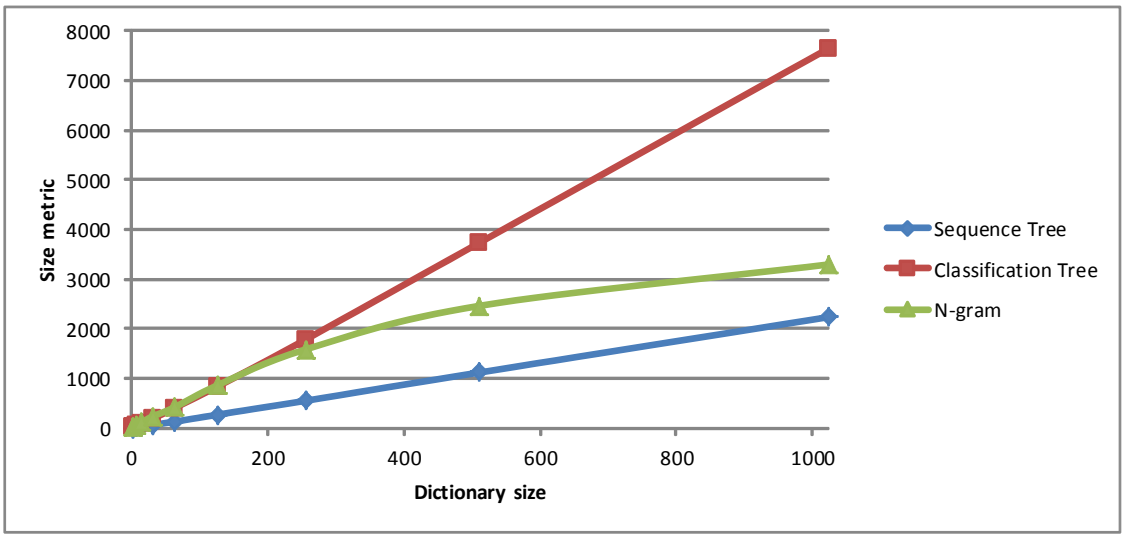

Figure 7.9: Classifier spatial requirements with increasing dictionary size

As the first test stops at a very small dictionary size we have performed another test with only sequence trees and n-gram prediction. The second test is depicted in Figure 7.10. As it can be seen in both figures, Sequence trees and n-grams are considerably smaller than classification trees on our test data. Classification trees does not seem to handle the sequential data very well as the size rises drastically at each dictionary size increase.

The other two algorithms can handle larger dictionaries much better. Both predictors can to some extent reuse data to reduce the impact on size when new sequences are added. Sequence trees reuse already created nodes when similar words are found; two words such as "phone" and "phony" may reuse the nodes p, h, o, and n. The n-gram predictor can also reuse already found n-grams; n-grams such as $(\{a, b, c\}, b, 1)$ $(\{a, b, c\}, c, 1)$ could reuse the sequence $\{a, b, c\}$ instead of duplicating it for each n-gram. These properties limit the growth of both sequence trees and n-gram predictors significantly.

In the case of n-grams there need not be many words before the growth decreases, as there are only so many three letter combinations to be found in common words. As an can be seen from Figure 7.10 n-grams only seem to show a logarithmic growth.

We expected the growth to decrease over time with sequence trees as happens

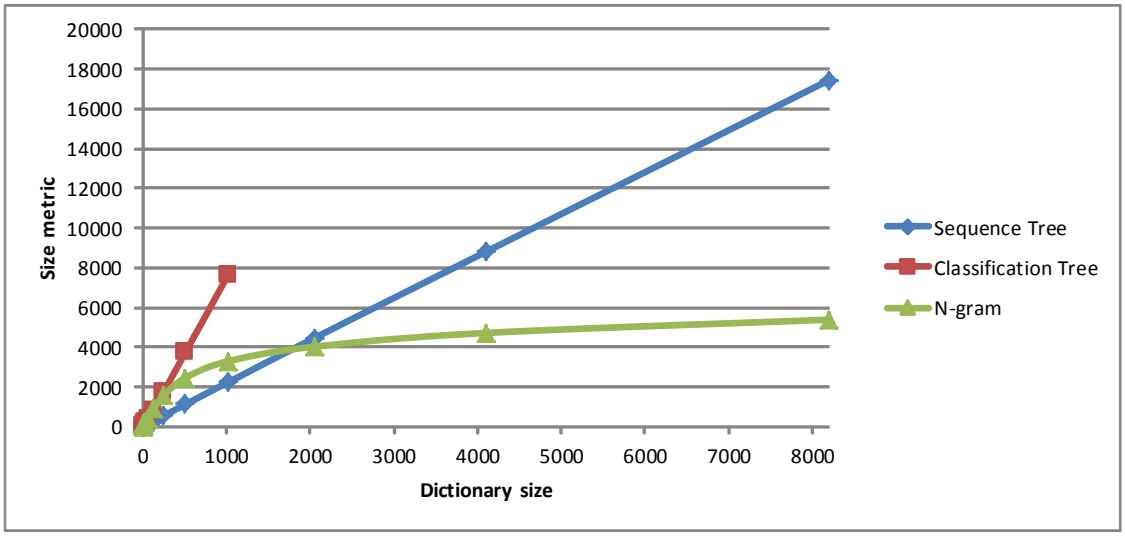

Figure 7.10: Classifier spatial requirements with larger dictionary sizes

with n-grams, however the training data may not contain enough repetitions to make this happen for sequence trees. However the growth would always grow by at least one node for each new sequence in sequence trees if that label has not been seen before. Because of this the size growth of sequence trees will never be less than a linear growth.

We can conclude that even though sequence trees have been optimized by collapsing we may need to use lossy optimizations such as the described pruning algorithm to make it small enough to compete with n-grams in size.

In the next section we will perform a test of the prediction accuracy of the three predictors.

7.6 Accuracy Testing

We saw in the last test that sequence trees perform much better than classification trees in terms of spatial requirements, but worse than n-gram predictors. In this section we will look on the prediction accuracy of the three different prediction methods. The accuracy of the methods is of course paramount to being good predictors. However in this test we will also look into how small partial sequences we need to do a correct prediction, as the sooner each method can predict a word the better the predictor. The test is performed on partial words as described in Section 7.1 with 10 partitions. The dictionary size is 1024 words, limited by classification trees. The results of this test can be seen in Figure 7.11.

As expected the accuracy is very low on small percentages of the words. At 50% of the full word written a large jump in accuracy is seen, where the accuracy is 67% for classification trees and 63% for sequence trees while the accuracy of n-grams cannot follow the two other methods reaching only 55% accuracy at this point. The goal of the prediction methods is obviously to have a high accuracy with the shortest amount of the original word.

When the full words are written we see that sequence trees will in 97% of the

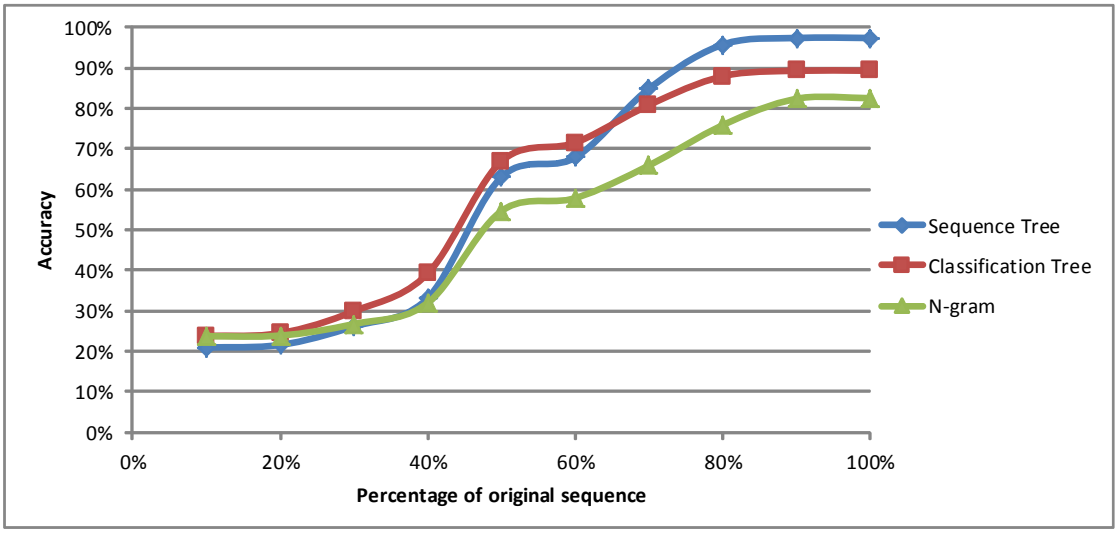

Figure 7.11: The accuracy of each method with incremental partitioning of words

cases suggest the correct words where classification trees will only recognize the full words in 89% of the cases and 82% for n-grams. This is interesting in the way that each predictor should be able to correctly distinguish fully written words. Most of the inaccuracy in the case of 100% word length consists of words that have the same representation of numberfied words as discussed in section 7.2; in this case it is impossible without contextual knowledge to determine which of the possible labels to select correctly in the case of sequence trees.

We can see in this test that sequence trees offer an acceptable level of prediction based on incomplete words, with halfway written words predicted with 67% certainty in the suggestion list. In the terms of predictive power the sequence tree algorithm seems very good as a method for preditive text. In the next section we will begin to examine another aspect of prediction in the domain of predictive text; the amount of KSPC we can attain by using the three prediction methods.

7.7 KSPC Testing

In this section we will test the keystrokes per character we can achieve with the three prediction methods sequence trees, classification trees, and n-gram prediction. First we will discuss how we measure the KSPC during our tests. Then we will test the KSPC for each predictor method on a single text corpus, and lastly we will use different corpus to see whether the selection of words in the training data has an influence on our methods' effectiveness.

In the previous accuracy test only the most probable prediction were evaluated when determining whether a prediction was correct or incorrect. When we measure KSPC we will have a number of predictions; we call these suggestions. We define a correct prediction when the correct word appears in the suggestion list.

When using the selection list and calculating KSPC we need to define how we measure the extra keystrokes from selecting words in the suggestion list. To recap

the original equation to calculate KSPC from Section 1.2 see Equation 7.1:

$$KSPC = \frac{\sum_w (K_w * F_w)}{\sum_w (C_w * F_w)} \tag{7.1}$$

Here K_w is the keystrokes for word w, F_w is the frequency of w, and C_w is the number of characters in w. All words have an additional space character appended in the end.

In his paper MacKenzie[11] measures the number of keystrokes for predictive text by the number of keystrokes to make the correct word first appear as a suggestion and then how many keystrokes used to select it in the list. He also measures the KSPC for a PDA, where the user can select any suggestion in the list by a single operation instead of moving a cursor up and down in the list. This form of word selection is equivalent to what is used in today's touch screen phones, and we will refer to this input method as touch selection.

In the KSPC tests we operate with two different modes of selecting words inspired by MacKenzie; keypad and touch selection. Each represents a different approach found on various mobile devices for word completion.

Keypad selection is where a list of suggestions appears, and the user must use a keystroke to go up or down the list and finally use a keystroke to select the word in the list. Touch selection is seen on modern smart phones etc. where the user can select any suggestion in the list with one keystroke. We calculate these two modifications of the original KSPC equation as shown in Equation 7.2 for keypad selection, and touch selection is calculated as shown in Equation 7.3.

$$KSPC_{Keypad} = \frac{\sum_w ((K_w + positionInList + 1) * F_w)}{\sum_w (C_w * F_w)} \tag{7.2}$$

$$KSPC_{Touch} = \frac{\sum_w ((K_w + 1) * F_w)}{\sum_w (C_w * F_w)} \tag{7.3}$$

Now the general setting for the KSPC test is detailed we still need to find a suitable number of predictions to find for each word to fill the suggestion list. In the following we will experiment with finding a good number of suggestions.

7.7.1 KSPC Suggestion Size

As discussed above we need to find a size for the suggestion list. The suggestion list defines how many possible solutions are shown to the user when the methods do a prediction. The size of the suggestion list affects how accurate a predictor has to be. A very large number of suggestions will lower the KSPC as a predictor can give multiple answers for a prediction, in effect increasing the chance of a correct prediction.

A very large suggestion list is detrimental for the KSPC for keypad selection however, as there may be many keystrokes needed to move through a large list effectively removing any advantage of predicting the word instead of writing it using multi-tapping described in section 1.3. The KSPC of touch selection does not degrade as keypad selection does as the number of suggestions increases, as regardless

of the number of suggestions it only requires one keystroke to select any suggestion. However we expect that at some point we will not gain enough benefit from increasing the suggestion size as the suggestions at some point will have very low probability and require additional computation time.

We would therefor like to know what size the suggestion list should have in the KSPC. We have experimented with predicting with different sizes of the suggestion list and calculated the average KSPC needed to complete a word. The result of this test can be seen in Figure 7.12.

Of the two input methods keypad selection has the highest KSPC. The keypad input mode needs to use keystrokes to move through the list and select the chosen word, so the additional keystrokes made will negate much of the benefit from a larger suggestion list. The KSPC of the touch input method falls drastically as the list size increase because every word in the list is accessible and larger suggestion lists can only improve the KSPC.

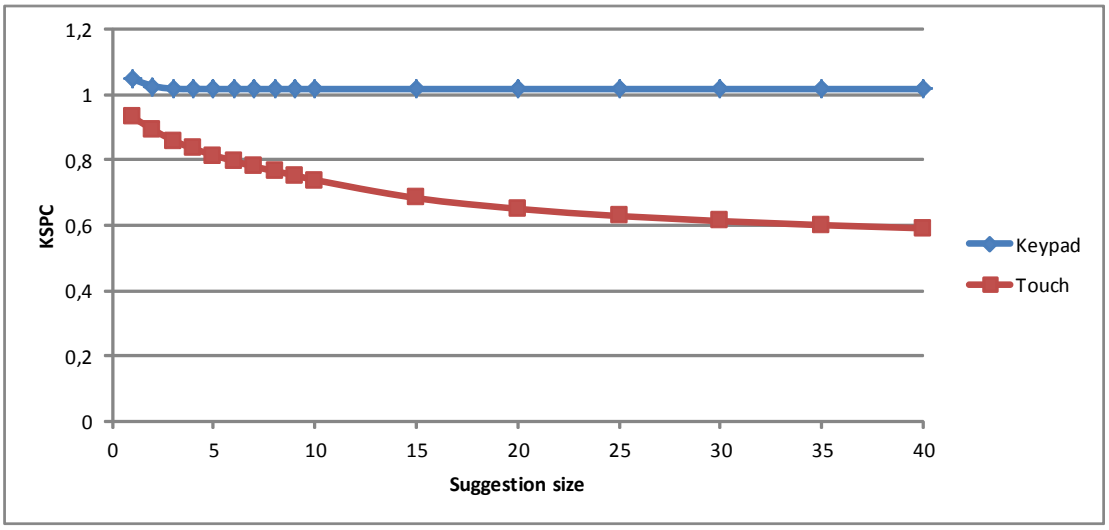

Figure 7.12: The suggestion list size versus the average KSPC

The KSPC falls quickly as the suggestion list is expanded to the size of up to five suggestions. At higher numbers of suggestions the KSPC declines at a slower rate, as the additional suggestions become more and more unlikely. At each increase of the suggestion list there is also a small slowdown of the prediction process, which makes very large suggestion lists less attractive. An additional point to note is that in real case scenarios it may not be possible to display a very large amount of suggestions on a mobile device.

Therefore we choose to use a suggestion size of 5 for the KSPC test in the next section, as this seems as the best compromise between low overhead from additional prediction candidates and the advantage of having several candidates. In the next section we will use the results from this test to examine in detail the KSPC that we can attain from our three prediction methods.

7.7.2 Keypad and touch KSPC comparison

As we have the found a suggestion list size to use we can now continue to experiment to attain a good KSPC using Sequence trees, classification trees and n-grams. In this test we use the test approach outlined in Section 7.1 where we divide words into 10 partitions. As discussed earlier in the spatial requirement test in Section 7.5, classification trees cannot manage dictionaries larger than 1024 words.

As a result thereof we have done two KSPC tests, one constrained to a dictionary size of 1024 where we look at both keypad and touch selection, and a second with 32785 words to see if KSPC is stable even with a larger dictionary. In the second we only consider touch selection as the input method as we do not want to compare the selection method but instead how the dictionary sizes affect KSPC. Both use a suggestion list size of 5 words as found previously. In Figure 7.13 the average KSPC can be seen for each prediction method for dictionary size 1024. The KSPC results for sequence trees and n-gram predictors on the larger dictionary can be seen in Figure 7.14

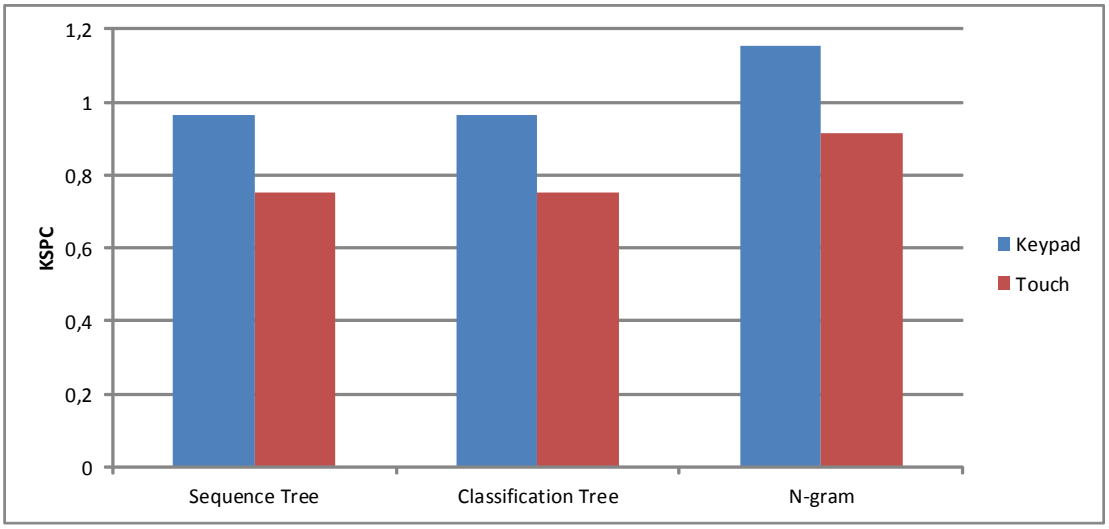

Figure 7.13: Average KSPC

Sequence trees and classification trees have a KSPC below one for both keypad and touch selection. N-grams for keypad selection lie just above 1 KSPC, but they can reduce the KSPC to under one using touch selection. As can seen in Figure 7.13 using keypad or touch selection makes a significant difference in the KSPC attainable. When we test sequence trees and the n-gram predictor on a much larger dictionary we can see the KSPC rises only slightly; this is to be expected as the ambiguity rises as more words are included in both methods. The larger dictionary is roughly 3199% larger than the small dictionary, however the increase in size only incurs a 4% and 9% increse in KSPC for sequence trees and n-gram prediction respectively.

Overall the KSPC is quite acceptable; recall that the commercial T9 method for predictive text had a KSPC of 1.0075 calculated by MacKenzie [11]. Additionally a KSPC around one is very good considering that the methods both need to predict and disambiguate keypad numbers into real words. As a final comparison we have

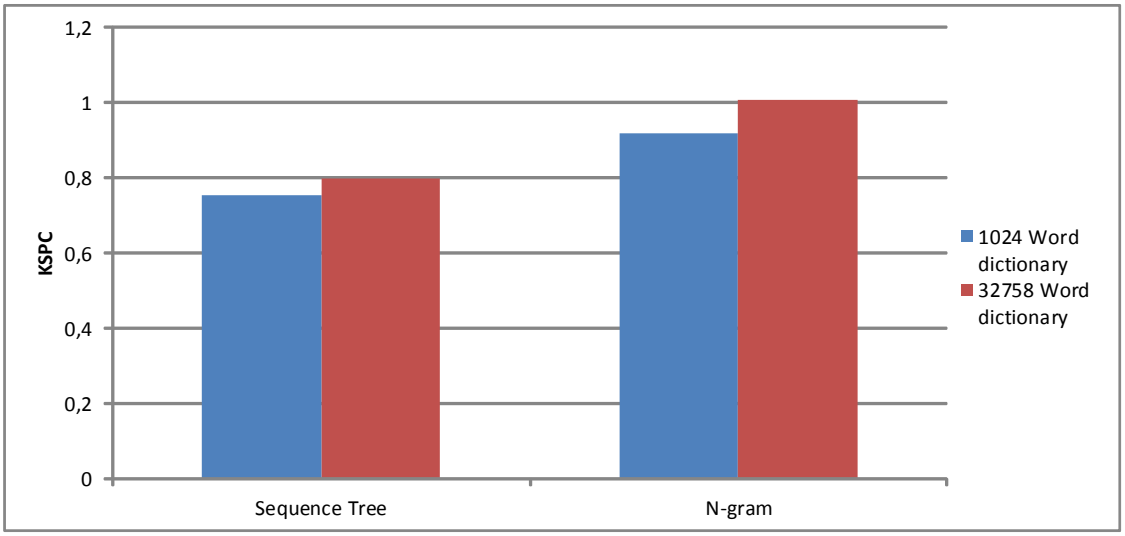

Figure 7.14: Sequence tree and n-gram predictor KSPC test on large dictionary sizes

calculated the multi-tap KSPC for the same dictionary used for the KSPC test which resulted in a KSPC of 2.35.

To make sure that the prediction algorithms perform equally well in terms of KSPC regardless of the test data we will perform an additional experiment using different dictionaries and then calculate the KSPC for sequence trees and n-grams.

7.7.3 Testing on Different Dictionaries

In the previous tests we have only used a single dictionary. In this test we would like to see whether we can get as good KSPC results from a number of different dictionaries, to see whether sequence trees can reduce KSPC in a general domain rather than only in our full test dictionary.

Recall that we in Section 7.2 listed four different corpus: "all", "context", "demographic" and "written". "context" contains context governed meeting words, "demographic" contains words from regular text message conversations, "written" contains words from popular written texts, and the "all" corpus contains all words from the three corpus.

Each of these four corpus have been used as training data for sequence trees as well as n-grams, and the resulting KSPC from each of these tests can be seen in Figure 7.15. Since the tests were performed on dictionaries of 32785 as in the touch and keypad comparison tests classification trees have been left out as they could not handle more than 1024 word dictionaries.

As can be seen the KSPC for sequence trees only increases for the "demographic" corpus, and only by 0.05 or less. KSPC vary slightly more for the n-gram predictor as each corpus results in different KSPC, but again the KSPC vary only by less than 0.05. So overall these different corpus do not have much impact on the KSPC, which indicates that sequence trees can be used for prediction within different domains of predictive text without an adverse effect on prediction accuracy.

Following this comparison of the different predictors' KSPC, as well as selection

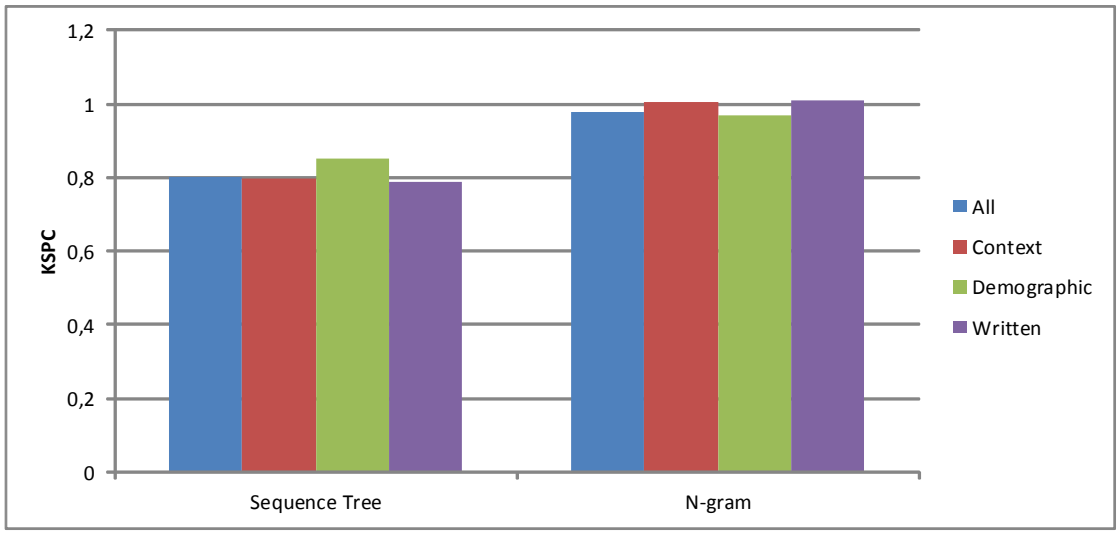

Figure 7.15: KSPC for sequence trees and n-grams tested on a variety of different dictionaries

of suggestion size and prediction on different predictive text corpus we now summarize over the test chapter, refreshing the important results found for sequence trees.

7.8 Summary

In this section we give a short summary of all the tests conducted throughout the chapter.

- **Sequence tree optimizations**
  The different optimization techniques to improve on sequence tree were tested. We examined the results of collapsing, pruning and clustering techniques to reduce the spatial requirements, where both collapsing and pruning showed promising results, however clustering was not too successful in the domain of predictive text. Furthermore we looked at how sorting was able to improve prediction speed at the cost of more time to insert new data. Lastly we looked at distribution sampling to see if it could improve accuracy, the results however showed that it did not make any significant improvement on prediction accuracy.

- **Spatial Requirement testing**
  We then tested the spatial requirements for classification trees, n-grams and sequence trees. Classification trees showed severe limitations being only able to handle a dictionary size of a maximum of 1024 words before running out of memory. Both n-grams and sequence trees did not have these memory limitations and were able to be tested on much larger dictionaries. When comparing the methods classification trees had by far the largest memory requirement and showed a linear growth, sequence trees showed a similar linear

growth although at a lower rate. Sequence trees performed well, however n-grams showed only a logarithmic growth as the dictionary size grew resulting in a very good spatial performance compared to the other prediction methods.

- **Accuracy testing**
  The accuracy of the three methods was examined next by using partial sequences of the original sequences to test how well the methods could predict at different partition sizes of the sequences. Classification trees showed a small edge with partial sequences that only contained very small partitions of the original sequences, however as more of the sequences were given, the better sequences trees performed in comparison with the other two methods. N-grams showed decent results but the predictor was outperformed by both sequence trees and classification trees.

- **KSPC testing**
  Lastly we looked into KSPC as an indicator to measure the usefulness of the methods in the domain of predictive text on mobile devices. We looked at what suggestion size to use and settled on a suggestion list size of five, this being the most realistic value to use on mobile devices and the lowest value before increases in size gave diminishing returns on the reduction in KSPC. In the main test KSPC was calculated for all methods with keypad and touch selection. Classification trees and sequences showed almost identical results with n-grams producing noticeably worse KSPC. Because of spatial limitations of classification trees it was not possible to test it on larger dictionaries. N-grams and sequences trees were tested with larger and different dictionaries to verify that results were consistent and independent of the data used to train on. The test showed only a small increase in KSPC as dictionary size grew and only minuscule fluctuations in KSPC with regards to different dictionaries.

Following testing of our sequence trees predictor method we next conclude on the project, summarizing the contents of the thesis as well as the key results of sequence trees and suggestions for further improvement and usage of our new prediction method.

CHAPTER
EIGHT

CONCLUSION

In this chapter we will give a summary of our project and conclude on our new sequence prediction method called sequence trees within the context of predictive text by comparing it to existing classification trees and n-gram predictors. After this we will discuss what further work could be possible on the sequence tree predictor.

8.1 Project Overview

Initially in this report we provided basic information on the domain of predictive text and ambiguous input inherent in mobile devices, as well as providing definitions of sequence prediction and event sequences. Additionally we described classification trees and n-gram predictors, two existing methods used for predictive text.

We next defined sequence trees within the general domain of sequence prediction, including how sequence trees are constructed as a series of sequence insertions, as well as how sequence trees can be used for prediction by returning a predicted label for a given input sequence.

We also explained how sequence trees can be used to model n-grams, and proved this using mathematical induction.

We then used a corpus of English words as training and testing data for running a series of tests on sequence trees and its n-gram and classification tree counterparts, looking at their spatial requirements, prediction accuracy and average keystrokes per character required for completing words in a predictive text system.

8.2 Conclusion

As stated in the project goals we have looked at the following properties of sequences trees:

- **Prediction accuracy**
  We have achieved a high prediction accuracy with sequence trees; when we know more than half of a sequence we can predict the correct sequence with high accuracy. Sequence trees showed itself to be better at predictive text than n-grams and just as accurate as classification trees.

- **Space requirements of the predictor**
  Sequence trees use a limited amount of space, and this grows only linearly as the training data increases in size. It uses much less space than classification trees but more than n-gram predictors, however using the proposed pruning technique we can reduce the size of sequence trees greatly without losing much accuracy.

- **Average number of KSPC needed to complete a range of words on a simulated mobile phone keypad**
  We have showed that we can reduce KSPC with sequence trees below what is possible on a qwerty keyboard without prediction, thus we can in average predict words and speed up writing on mobile devices significantly. Sequence trees and classification trees performed very well however as classification trees could only be trained on very small dictionaries the method is not a valid method. n-grams performed worse in terms of KSPC than the other predictors, however it can as sequence trees be trained on very large dictionaries.

Among the sequence tree optimizations, clustering and some of the collapsing techniques were not in use in the final tests, as they did not contribute positively to the size or accuracy of the sequence trees in the domain of predictive text. We do however believe in other domains that these optimizations may be useful.

Thus we can conclude that we have developed a sequence prediction method that can recognize and predict sequences with high accuracy and low spatial requirements in the domain of predictive text.

8.3 Further Work

The prediction method we described in this report works very well when predicting words, however further improvements can still be made. We have reduced the size of sequence trees using our optimizations, however this size reduction could be further researched to make sequence trees even more viable in the context of predictive text and mobile devices.

So far we have only tested sequence trees in the context of predictive text however we believe that the method might be possible to extend in different domains involving sequential data.

Sequence trees could also be made customizable to the user; as it is quite simple to insert new sequences and update the frequencies of existing sequences, it is possible to add new words to the sequences tree or change the frequencies on the edges as the user writes messages using sequence trees.

Another topic of interest is extending the method to include more contextual data to the predictions. In predictive text this could be the previous messages a mobile device has received, such that when a user receives a message on a specific topic the probabilities of some paths of the tree can change when he is writing a response depending on the content of the previous message.

Some of the erroneous predictions were caused by identical sequences with two or more labels. By having additional contextual information about words such as

that from previous messages it may also be possible to better determine which label to return when predicting in this situation.

It could also be possible to create a multilayer sequence tree, such that the upper layer predicts sentences and the lower layer predicts words. This allows for a fuller text completion system and potentially even less KSPC needed for completing entire sentences or messages.

All these new ideas however requires some time to test and implement as some changes in the basic algorithms must be made to incorporate things such as contextual information in a sequence tree. All in all we believe that the sequence prediction method may be further developed, but as we have seen it can already compete with other predictors such as classification trees and n-gram predictors in size and accuracy.

BIBLIOGRAPHY

[1] J. Anderson. *Learning and Memory.* John Wiley & Sons, 1995.

[2] Various authors. Presage `http://presage.sourceforge.net/`. Web, 2011.

[3] Steffen Bickel, Peter Haider, and Tobias Scheffer. Predicting sentences using n-gram language models. *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2005.

[4] Ezra Black, Fred Jelinek, John Lafferty, Robert Mercer, and Salim Roukos. Decision tree models applied to the labeling of text with parts-of-speech. In *Proceedings of the workshop on Speech and Natural Language*, pages 117–121, 1992.

[5] Henrik O. Sørensen Peder S. Sørensen Johannes G. Nielsen Jakob S. Knudsen Frederik K. Frandsen, Mikkel F. Hansen. Predicting player strategies in real time strategy games. University Project at Aalborg University, 2010.

[6] Jon Hasselgren, Erik Montnemery, Pierre Nugues, and Markus Svensson. HMS: A Predictive Text Entry Method Using Bigrams. In *Proceedings of the Workshop on Language Modeling for Text Entry Methods, 10th Conference of the European Chapter of the Association of Computational Linguistics*, pages 43–49. Association for Computational Linguistics, 2003.

[7] J. Häkkinen and Jilei Tian. N-gram and decision tree based language identification for written words. In *Automatic Speech Recognition and Understanding, 2001. ASRU '01. IEEE Workshop on*, pages 335–338, 2001.

[8] Manfred Jaeger. Mi course lecture notes. `https://intranet.cs.aau.dk/fileadmin/user_upload/Education/Courses/2009/MI/Slides/mi-E09-08.pdf`, 2009.

[9] Finn V. Jensen and Thomas D. Nielsen. *Bayesian Networks and Decision Graphs.* Springer, 2007.

[10] Ron Kohavi and Ross Quinlan. Decision tree discovery. In *Handbook of data mining and knowledge discovery*, pages 267–276. University Press, 1999.

[11] I. Scott MacKenzie. Kspc (keystrokes per character) as a characteristic of text entry techniques. *Mobile HCI '02 Proceedings of the 4th International Symposium on Mobile Human-Computer Interaction*, 2002.

[12] Scott I. Mackenzie, Hedy Kober, Derek Smith, Terry Jones, and Eugene Skepner. LetterWise: prefix-based disambiguation for mobile text input. In *UIST*, pages 111–120, 2001.

[13] David M. Magerman. Statistical decision-tree models for parsing. In *In Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics*, pages 276–283, 1995.

[14] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[15] Oxford University Computing Services on behalf of the BNC Consortium. The british national corpus `http://www.kilgarriff.co.uk/BNClists/all.num.gz`. Web, 2011.

[16] Richard M. Stallman. *GNU Emacs Manual: For Version 22, 16th edition*. Free Software Foundation, 2007.

[17] R. Sun, E. Merill, and T. Peterson. From implicit skills to explicit knowledge: A bottom-up model of skill learning. *Cognitive Science, volume 25 no 2, pages 203-244*, 2001.

[18] Ron Sun. Introduction to sequence learning. In Ron Sun and C. Giles, editors, *Sequence Learning*, Lecture Notes in Computer Science, pages 1–10. Springer Berlin / Heidelberg, 2001.

[19] Ron Sun and C. Lee Giles. Sequence learning: From recognition and prediction to sequential decision making. *IEEEIntelligent Systems, p67-70*, Jul-Aug 2001.

[20] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley, 2006.