Institut für Parallele und Verteilte Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3212

Efficient Prediction Model Management in Mobile Systems

Jörg Belz

Studiengang:	Informatik
Prüfer:	Prof. Dr. K. Rothermel
Betreuer:	Dipl.-Inf. Stefan Föll
begonnen am:	13. Juli 2011
beendet am:	12. Januar 2012
CR-Klassifikation:	G.1.6, C.2.4

Contents

List of Figures

Abstract

With the advent of affordable mobile devices such as smartphones and tablets, the vision of Pervasive Computing has made a big step closer to becoming reality. In order to become truly ubiquitous and seamlessly integrate into everyday life, the design of context-aware applications is essential. Using contextual information obtained for example from the device's sensors such as motion sensors and gps receiver, context-aware applications can adapt their behavior depending on the environment the user is in. In some scenarios, context aware applications can also benefit from knowledge about future contexts. This necessitates the use of a context prediction model. We examine a social network scenario where in addition, the context in question is originally being acquired on another user's device. In this scenario, the prediction model could for example be used to predict the next location or activity of a friend. Prior to that, the prediction model needs to be distributed to and stored on the mobile device running the application. Both high transfer cost and limited space make it imperative to produce small prediction models which still predict the context considerably well. In this thesis, we examined methods to compress Markov-based prediction models of higher order in a lossless and lossy fashion and evaluated these methods on real world and generated data. Our evaluation showed clearly that the compression mechanisms introduced can be successfully applied to significantly reduce the size of the prediction models with only a minor impact on prediction performance.

1 Introduction

1.1 Motivation

The term 'context' is defined by the Merriam Webster dictionary as 'the parts of a discourse that surround a word or passage and can throw light on its meaning' [MW]. Therefore, context is something that is not being explicitly expressed in any way, but nevertheless useful for the situation at hand because of the additional information it provides. Humans act context-aware every day, often without noticing. For example, when standing at the checkout in a shop, there is no need to vocally express the intention of purchasing the items carried. The context, which in this example is represented by the location, is already sufficient. So context awareness can greatly simplify but also enhance human-human interaction.

The field of Context Aware Computing aims to transfer this to the interaction between humans and computers by providing context information to applications. Traditionally, manual input by the user is the only information source and many applications can benefit from the additional information a context contains. For example, the human interaction with an application can be improved by emphasizing information depending on the current location [SAW94]. Jakob Bardram [Bar04] describes how context awareness can be incorporated into an application showing medical records on a screen attached to the hospital bed. It chooses the information to display depending on the location of the nurse, the bed and the patient. Other uses of contextual information are the adaption of the applications behavior depending on the user and the discovery and usage of resources within the same context [Pas98][ST94]. An application can for instance automatically choose a printer in the same room to print a document [SAW94]. Furthermore, the context can be augmented by information related to the context and displayed to the user [Pas98], which leads to the concept of augmented reality [SMR+97].

Contextual information can therefore enhance an application in numerous ways. In some scenarios, not only information about the current context, but also about future contexts may be beneficial. For example, in a smart building the knowledge about the future location of a person can be used to prepare the room the person will enter next [PBTU]. User context prediction can also be employed in order to predict network availability [RO08]. In a social network, a notification message can be published to a user when a friend is expected to arrive at his location soon. Furthermore, context prediction can also be used to reduce energy consumption in a social network scenario involving mobile devices [Fin11]. However, context prediction requires the storage of a prediction model which may exceed the memory designated to the application. This in particular applies to mobile devices which only have limited space available. Since mobile devices play an important role in context-aware

systems due to their mobility and multitude of sensors, the question arises how to store these prediction models efficiently. In this thesis, we examine methods to compress prediction models in a lossless and lossy fashion and evaluate the methods on real world and generated data.

1.2 Outline

In the following chapter, we will briefly discuss related work. Afterwards we will describe the system model used and give a problem specification. In chapter 4 we will introduce the prediction models for which lossless and lossy compression methods will be proposed in chapters 5 and 6. Then we will describe how the compression can be applied in our scenario. An evaluation of the predictors will be conducted in chapter 8 before we summarize the results and give an outlook in the final chapter.

2 Related Work

The term of context-aware computing was introduced by Schilit and Theimer [ST94] in 1994. They define context-aware computing as "the ability of a mobile user's applications to discover and react to changes in the environment they are situated in". Context awareness is an essential part of Pervasive Computing (Ubiquitous Computing), first described by Weiser in his vision 'The computer for the 21st century' [Wei91]. He presents a future scenario where modern information technology fades into background, comparable to the technology of writing which became ubiquitous in modern life.

Even though the notion of a context plays an essential role in context-aware systems, no universal definition of context exists in the literature. Schilit et al. [SAW94] identify three important aspects of contexts based on the location: The location itself and people and resources nearby. Furthermore, they include information such as lighting, noise level, network connectivity and the social environment into their idea of a context. The notion of context by Schmidt et al. [SBG98] is that of a set of features describing a situation and the environment. Day and Abowd [DA00] provide a more general definition of context:

> "Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves."

Zimmermann et al. [ZL07] extend this definition by categorizing the information a context holds into individuality, activity, location, time and relations. The context is acquired using data provided by sensors. Indulska et al. [IS03] classify sensors into physical, virtual and logical sensors. Typical physical sensors include sensors for location, audio, video and acceleration [BDR07]. Virtual sensors obtain information from software applications or services, for example by using information the user previously provided to the application. Logical sensors infer information by using data provided by physical and virtual sensors. Such sensor signals typically represent low level context information which needs to be processed to obtain higher level contexts [Mö3]. This can be accomplished for example by using Naive Bayes [KKP+03], nearest neighbor, decision tree or decision table classifiers [BI04][IBB03].

Existing work on context prediction mainly focused on next location prediction. Song et al. [SKJH04] evaluated Markov and LZ (Ziv Lempel [ZL78]) based prediction models on mobility patterns of college students. Their evaluation results show that the Markov predictor of order 2 performed best overall. For higher orders however, the predictor performance decreased. The prediction accuracy for order 2 could be slightly improved by employing a fallback mechanism that allowed the predictor to make predictions for states not seen before by falling back to a lower order state. Katsaros and Manolopoulos [KM05] compared various

predictors: Prediction by partial matching (PPM [CIW84]), LZ78 ([ZL78]), a prediction suffix tree predictor (PST [RST96]) and the context tree weighting method (CTW [WST95]). Then they proposed a new method called suffix tree based predictor (STP) which is a generalization of PPM and LZ78. The STP performance was however not evaluated in comparison with other predictors. Begleiter et al. [BEYY04] evaluated several variable order predictors on different sample datasets. The CTW and PPM predictors performed best with no statistically significant difference between the results.

Rathnayake and Ott [RO08] use a semi Markov model of higher order to predict the average WLAN availability within the next 5 minutes based on the current context information. The context included the time of day, current WLAN availability, LAN availability, power on AC, number of bluetooth devices nearby and the GSM cell id. They evaluated the prediction model for four users that were using a device logging the aforementioned context information. The evaluation showed that actual average WLAN availability differed by 26% from the predicted availability during periods when the availability was fluctuating. The prediction could be improved for 3 of the 4 users by increasing the order from 1 to 2 and stays roughly the same for the 4th user. Musolesi et al. [MPF$^+$] discuss efficient updating strategies for Markov based prediction models in a distributed environment. They motivate a client based storage of prediction models by the changing availability of network connectivity and high cost of transferring sensed information in mobile networks.

3 System Model

In this chapter we will describe the network model that we assume and briefly give some definitions required in the following chapters.

3.1 Network Model

Our scenario is composed of a publisher-subscriber network in which the communication takes place through a server. Publishers disseminate content which is received by subscribers that are subscribed to them. In our setting, publishers represent people in a social network that produce new contexts and subscribers represent people that want to receive updates on this context. They are in a $n : m$ relation, so one subscriber may be subscribed to several publishers and one publisher may have several subscribers. An individual can subscribe to a publisher's context updates by sending a subscription message with the publisher id to the server which manages the publisher-subscriber relations. Messages from the publisher with several recipients can be relayed through the server so the publisher needs to send only one message which is then distributed by the server. So in our scenario the server merely acts as an intermediate that keeps the publisher from having to send a message to each single subscriber and to manage the subscriptions himself.

The subscribers also store a prediction model for each publisher they are subscribed to. The prediction models are either transferred at some point from the publisher to the consumer or generated on the consumer device. These prediction models can be used by a context-aware application running on the subscriber's device to predict the future context of a publisher and adapt accordingly.

3.2 Definitions

We define a context σ of a publisher as any information that describes the *current and future* situation of the publisher and that is relevant to the interaction between his subscribers and their applications. For example. σ can describe the future activity or location of the publisher. This represents a slight modification of the definition proposed by Dey and Abowd [DAoo] that not explicitly states that the future context may be relevant to the user but also not contains any reference to time whatsoever. Therefore, our definition is still coherent with the definition by Dey andAbowd. The set of all such contexts σ is denoted by $\mathcal{C}$.

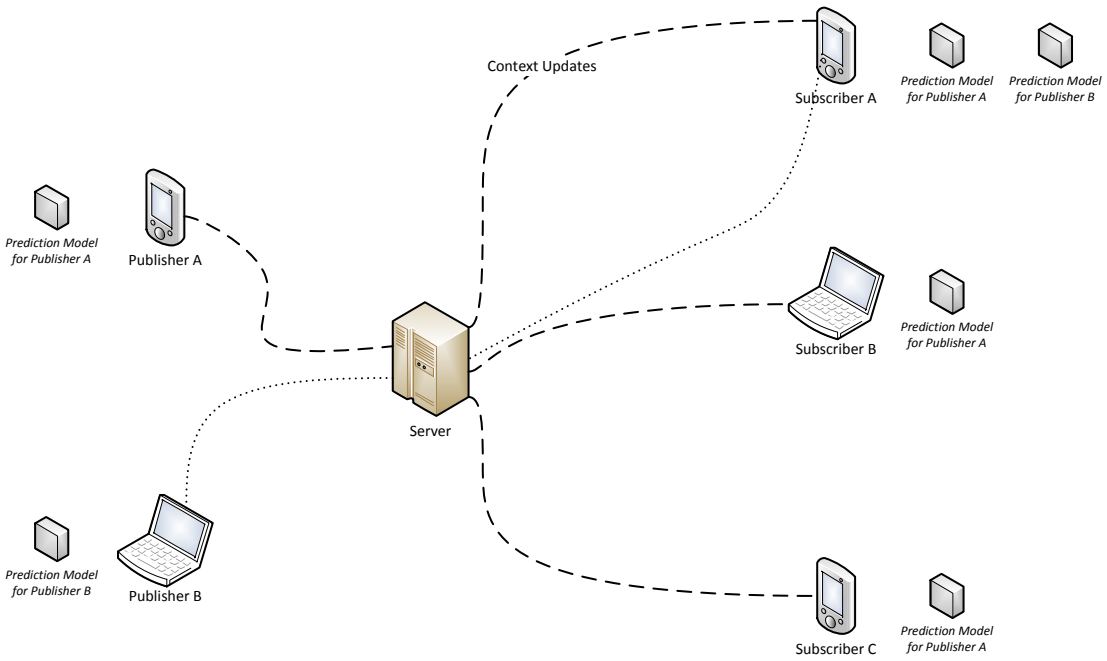

Figure 3.1: Network structure

The context is obtained in periodic intervals by the application for example by classifying sensor readings. Whenever a publisher's context changes, that e.g. the information relevant to the subscribers changes, he sends an update message containing the new context to the server which in turn relays the message to all corresponding subscribers (push service). We call such a change from context σ to context σ' context transition and denote it by their concatenation $\sigma\sigma'$. Multiple context transitions can likewise be expressed by a sequence $\sigma_1\sigma_2\cdots\sigma_k$. The history of context transitions that lead to the current context is called a state. A state of order k contains the last k context transitions $\sigma_1\sigma_2\cdots\sigma_k$ with σ_k being the current context. The state of order 0 contains no context information and is therefore called empty state $\perp$. A state is denoted by a $s \in \mathcal{S}_\mathcal{C}$ and the order of a state by $|s|$. $\mathcal{S}_\mathcal{C}$ is the unbounded set of all concatenations consisting of contexts from the set $\mathcal{C}$. Furthermore, we define $\mathcal{S}_{k,\mathcal{C}} = \{s \in \mathcal{S}_\mathcal{C} \big| \, |s| = k\}$, $\mathcal{S}_{\leq k,\mathcal{C}} = \{s \in \mathcal{S}_\mathcal{C} \big| \, |s| \leq k\}$ and $\mathcal{S}_{>k,\mathcal{C}} = \{s \in \mathcal{S}_\mathcal{C} \big| \, |s| > k\}$.

A subscriber is however not only interested in receiving new contexts as they occur, but also in future context transitions. Therefore, a subscriber holds a context predictor for each publisher he is subscribed to. The predictor is represented formally by a prediction function f which maps a state to the predicted context:

$$f : \mathcal{S}_\mathcal{C} \to \mathcal{C} \cup \sigma_\perp$$

The prediction function may not hold a prediction for every $s \in \mathcal{S}_\mathcal{C}$, for example because some predictions were removed due to compression. We indicate this case by the context $\sigma_\perp$. The size of a prediction function is determined by the number of contexts required to

store each state for which a prediction exists ($\sigma \neq \sigma_\perp$) and the corresponding predictions themselves.

In the next chapter, we will introduce Markov-based predictors which can be used to generate and store a prediction function. We will see that these predictors are difficult to use in our scenario because they often require a large amount of space to be stored.

4 Markov Prediction

A model widely used for prediction of complex systems is the Markov chain model. We will focus our prediction efforts on predictors based on this model. In the following sections, we will first provide a brief introduction to the Markov model. Then we will describe how this model can be used for context prediction.

4.1 Markov Models

4.1.1 Markov Property

A basic concept of Markov models is the notion of a state. A state is a set of variables that can be used to describe the current situation of the system. When one of these variables changes, the state changes and the system is transitioning into a new state. A process thus can be described by a sequence of states $s_1 s_2 \cdots s_k$. The aim of the Markov model is to predict the state that the current state will transition into based only on the information provided by the current state. The assumption that the current state's transition probabilities do not depend on past states is also known as Markov property [RN10]. Formally, we can express the Markov property for a state represented by the discrete random variable X as

$$P(X_{t+1} = x_{t+1} | X_t = x_t, X_{t-1} = x_{t-1}, \ldots, X_0 = x_0) = P(X_{t+1} = x_{t+1} | X_t = x_t)$$

where $P(X = x)$ denotes the probability of X taking the value x. Processes that adhere to this constraint are also called memoryless processes since they do not 'remember' the states they were in before. This is a very restrictive assumption and not likely to be fulfilled in most systems. A way to circumvent this restriction is allowing states of higher order. In our setting, a state of higher order does not only contain information about the current context, but also information about past contexts the publisher has been in. A process based on higher order states is not memoryless in a strict sense because the states contain a sequence of past contexts. However, as long as the states' orders are bounded, the Markovian assumption is still satisfied [Fino7] [How71].

4.1.2 Transition Probability Matrix

A Markov model for n states is represented by a $n \times n$ transition probability matrix $\mathbf{P}$ which contains the transition probabilities $p(s_k|s_i)$. $p(s_k|s_i)$ denotes the probability of a transition

into state s_k when in state s_i. A state may also transition into itself, this is denoted by $p(s_i|s_i) > 0$.

$$\mathbf{P} = \begin{bmatrix} p(s_1|s_1) & p(s_2|s_1) & \cdots & p(s_n|s_1) \\ p(s_1|s_2) & p(s_2|s_2) & \cdots & p(s_n|s_2 \\ \vdots & & & \\ p(s_1|s_n) & p(s_2|s_n) & \cdots & p(s_n|s_n) \end{bmatrix}$$

Note that with higher order states, the matrix is very sparse because a transition triggered by a context change can only change the state in a specific way. We define the $suffix(s)$ of a state $s = \sigma_1 \cdots \sigma_k$ as the concatenation of the last $k-1$ contexts: $suffix(\sigma_1 \cdots \sigma_k) = \sigma_2 \cdots \sigma_k$. When a new context σ is observed while in state $s = \sigma_1 \cdots \sigma_k$, the state s transitions into a new state $s' = suffix(s)\sigma$. So basically the new state s' is obtained by adding the new context that triggered the transition as the current context, shifting all existing contexts to the past. The oldest known context is removed from the state because the order does not allow storing $k+1$ contexts. Therefore, for a state s there exist only $|\mathcal{C}|$ possible successor states in a higher order model. It is sufficient to provide a matrix which only stores the context transition probabilities $p(\sigma|s)$ from each state s to each context $\sigma \in \mathcal{C} = \{\sigma_1, \ldots, \sigma_j\}$:

$$\mathbf{P}' = \begin{bmatrix} p(\sigma_1|s_1) & p(\sigma_2|s_1) & \cdots & p(\sigma_j|s_1) \\ p(\sigma_1|s_2) & p(\sigma_2|s_2) & \cdots & p(\sigma_j|s_2) \\ \vdots & & & \\ p(\sigma_1|s_n) & p(\sigma_2|s_n) & \cdots & p(\sigma_j|s_n) \end{bmatrix}$$

This modification reduces the number of entries from $n * n =$ to $n * |\mathcal{C}|$. Since for higher orders the number of states n is larger than the number of contexts, the matrix $\mathbf{P}'$ is considerably smaller than $\mathbf{P}$.

4.1.3 Learning

The transition probabilities $p(\sigma|s)$ in the transition probability matrix are generally unknown. However, they can be estimated using a training sequence $\mathcal{T} = \sigma_1\sigma_2 \cdots \sigma_N$ of $N-1$ context transitions. To this end, we determine the empirical transition probabilities $\hat{p}(\sigma|s)$ that we can use as estimators for $p(\sigma|s)$. Let $c(\sigma_1 \cdots \sigma_k)$ denote the count of occurences of the sequence $\sigma_1 \cdots \sigma_k$ in the training sequence. Then we define $\hat{p}(\sigma|s)$ as follows:

$$\hat{p}(\sigma|\sigma_1 \cdots \sigma_k) = \frac{c(\sigma_1 \cdots \sigma_k\sigma) + 1}{c(\sigma_1 \cdots \sigma_k) + |\mathcal{C}|}$$

The probabilities are adjusted to the total number of different contexts using a modified version of LaPlace's rule of succession. LaPlace's rule of succession states that after observing an event m times, the probability to observe it again is given by $\frac{m+1}{m+2}$ and therefore approaches 1 with an increasing number of samples m [Zab89]. Basically by using this rule we assume

to initially have uniformly distributed successor probabilities. That way we smooth the probabilities for states that rarely occur. For example given a state s for that $c(s) = 1$, e.g. s has been seen only once in the training sequence, the most likely successor would otherwise always have a probability of 1. While correct in a strict sense, the probability can not reflect the actual behavior due to the low sample size. For the empty state $\perp$, we estimate the probability by

$$\hat{p}(\sigma|\perp) = \frac{c(\sigma) + 1}{|\mathcal{T}| + |\mathcal{C}|}$$

where $|\mathcal{T}|$ denotes the length of the training sequence used to determine $c(\sigma)$. Thus, $\hat{p}(\sigma|\perp)$ is the smoothed unconstrained probability of σ occuring in the training sequence.

4.2 Prediction using Markov Models

Given the $p(\sigma|s)$, we define the predicted successor $succ(s)$ of a state s as the context for which $p(\sigma|s)$ reaches its maximum:

$$succ(s) = \operatorname*{argmax}_{\sigma} p(\sigma|s)$$

The predicted context is therefore the most likely successor context. If two such contexts exist, we can use a predefined ordering of contexts such as the order of their first occurence in the training sequence.

Consequently, the prediction function is simply defined as

$$f(s) = succ(s)$$

However, such an optimal prediction with respect to the training sequence can only be made if predictions for all states of all possible orders in the training sequence are stored in the transition probability matrix. As showed in the problem statement, the number of states increases exponentially with the order. This makes it infeasible to store such a prediction function. Therefore, the order must be restricted to an order n. It can either be restricted to states of exactly order n or to states of orders up to n. We denote predictors based on such restrictions fixed order predictor and full order predictor and will introduce them in the next sections.

4.2.1 Fixed Order Markov Predictor

A fixed order Markov predictor of order n can be defined using a transition probability matrix that contains only probabilities $p(\sigma|s)$ for states $s \in \mathcal{S}_{n,\mathcal{C}}$. The prediction function for the fixed order Markov predictor is defined as follows:

$$f(s) = \begin{cases} succ(s) & \text{if } s \in \mathcal{S}_{n,\mathcal{C}} \\ f(suffix_n(s)) & \text{if } s \in \mathcal{S}_{>n,\mathcal{C}} \\ \sigma_\perp & \text{else} \end{cases}$$

The fixed order Markov predictor only returns predictions for states greater than or equal to n. For states of order n, the prediction can be directly determined by the transition probabilities for that state. For states of order higher than n, we simply prune the state to the suffix of order n. This is done by the function $suffix_n(\sigma_1 \cdots \sigma_k) = \sigma_{k-n}\sigma_k$. In case the state s has a lower order, the fixed order predictor is not able to make a prediction which is indicated by the special context $\sigma_\perp$. A fixed order predictor of order 0 is defined for any state as it predicts the most likely context seen in the training sequence regardless of the current state.

4.2.2 Full Order Markov Predictor

The performance of the fixed Markov predictor depends heavily on the size of the training sequence. If a state s does not occur in the training sequence, no prediction can be made since no successor contexts are known for the state. In higher order models, this is aggravated by the fact that the higher the order, the larger the set of possible states for which predictions are to be found.

A possible solution is to include all states from order 0 up to a maximum order n. That way, if a prediction is not available for a given state, the prediction of the closest suffix for which a prediction exists can be used. A full order Markov predictor of order n combines $n+1$ fixed order Markov predictors so that it is defined for every state independent of whether it was seen in the training sequence or not.

Let f_k be a fixed order Markov predictor for order k. Then the full order Markov predictor is defined as follows:

$$f(s = \sigma_1 \cdots \sigma_k) = \begin{cases} f_k(\sigma_1 \cdots \sigma_k) & \text{if } f_k(s) \neq \sigma_\perp \\ f_{k-1}(\sigma_2 \cdots \sigma_k) & \text{else} \end{cases}$$

The prediction function of a full order Markov predictor is defined for any state because the prediction can always fall back to f_0 which simply predicts the most likely context in the training sequence. A full order Markov predictor of order n is trained by training $n+1$ fixed order Markov predictors for orders 0 to n.

4.2.3 Continuous Markov Predictor

Both Markov predictors introduced require a training sequence. However, in some applications, there is no training sequence available. To still make predictions, we can update

the transition probabilities gradually as new contexts are generated. Since the predictor is learning continuously, we call this predictor continuous Markov predictor. For the trained predictors, the transition probabilities were defined using the count function $c(s)$ which denotes the number of occurences of the state s in the training set. The continuous predictor updates this function with every context transition (algorithm 1). The updating not only needs to be done for the current state of order n, but also for all states that share a common suffix with s. To this end, we define the set of all suffixes of a state s with order n as

$$suffix^*(s) = \bigcup_{i=0\ldots n-1} suffix_i(s)$$

Next to s, these states are also affected when a transition from state s occurs. In particular, regardless of n, the transition is always counted for the empty state.

Algorithm 1: Learning of $c(s)$

```
initialize c(s) with 0 for all s
foreach transition from state s to context σ do
    c(sσ) = c(sσ) + 1
    foreach s' ∈ suffix*(s) do
        c(s'σ) = c(s'σ) + 1
    end
end
```

Like the full order Markov predictor, the continuous Markov predictor learns and stores $n+1$ fixed order Markov models. The prediction function is also defined equally to the full order Markov predictor.

$$f(s = \sigma_1 \cdots \sigma_k) = \begin{cases} f_k(\sigma_1 \cdots \sigma_k) & \text{if } f_k(s) \neq \sigma_\perp \\ f_{k-1}(\sigma_2 \cdots \sigma_k) & \text{else} \end{cases}$$

However, when the count function $c(s)$ changes, the most likely successor of a suffix of s may also change. Therefore, the prediction functions f_i are being updated after every transition. By not counting transitions for suffix states, a continuous Markov predictor corresponding to the fixed order Markov model can also be generated but we will focus on the continuous predictor storing predictions for all orders.

4.3 Problems of Markov Predictors

The Markov predictors described have some downsides which make it difficult to use them in our scenario. The Markov predictor of fixed order requires that each state it needs to make a prediction for has been seen in the training sequence. This is generally difficult to ensure for higher order models because the number of possible states, e.g. combinations of different contexts, is growing rapidly with the order. The full order Markov model does not

suffer from this disadvantage since it always can fall back to a lower order suffix state which is more likely to be seen. This however comes at the cost of having to store much more states than the fixed order model.

Even if we could find a way to train a predictor with most of the states, it is infeasible to store all of their predictions. For example, for order $n = 4$ and $|\mathcal{C}| = 30$ different contexts, the total number of possible states of order n is $|\mathcal{C}|^n = 810\,000$. Assuming 0.5 KB was necessary to store one context, storing context predictions for all these states and the states of order 4 themselves would require $5 * 810000 * 0.5$ KB $= 1978$ MB. Furthermore, a subscriber holds a prediction model for each publisher. Thus, for a large number of publishers storing the prediction models becomes even more problematic.

We assume the space available to store the prediction functions is constrained. This requires reducing the total size required by the prediction models by removing the number of states for which a prediction is stored. The goal is to develop compression mechanisms that have a low impact on the predictor performance. In the next chapter, we will introduce a lossless compression mechanism that does not change the prediction of the full order Markov model at all. Since this not always can reduce the size sufficiently, we will develop a lossy compression mechanism in chapter 6 that can compress prediction models to arbitrarily small sizes. Since this inevitably changes the prediction function, we will look into ways to compress the models as good as possible with respect to prediction performance.

5 Lossless Compression using Variable Order Models

A prediction model can be compressed by removing states and corresponding predictions. If possible, the removal should happen in a way so that the performance of the predictor does not suffer. Lossless compression requires that the compression must not change the prediction function of the model it compresses: Given a prediction model with a prediction function f, the lossless compressed model must represent a prediction function f' so that

$$\forall\, s \in \mathcal{S}_{\mathcal{C}} : f(s) \equiv f'(s)$$

Consequently, lossless compression has no impact on the predictions a predictor delivers for any state.

5.1 Variable Order Models

The problems described in section 4.3 are caused by the large number of states that a model of higher order contains. This particularly effects the full order Markov model which contains all states up to the specified order n. However, the underlying process may be of variable order, e.g. the next context depends on *at most* n last contexts. Hence, there are states s so that $f(s) = f(suffix(s))$. For such states, no prediction needs to be stored in the full order Markov model since the fallback already contains the correct prediction.

5.2 Prediction Suffix Trees

An approach for storing and learning predictions for a variable order model using a suffix tree was proposed by Ron et al. [RST96]. In a suffix tree, each edge of the tree is labelled with a $\sigma \in \mathcal{C}$ and each node has at most one child edge for each σ. The nodes are arranged in the tree so that the state s of a node can be obtained by concatenation of the edge labels seen on a walk from the node to the root. Hence, the deeper the node, the higher the order of its state. This is illustrated by the example in figure 5.1 for the contexts a and b. The root node represents the empty state and its children represent the states a and b. States of order 2 are represented by the children of a and b. Note that the states in the figure are given only in order to illustrate what states are assigned to the nodes. In an actual suffix tree, the states are implicitly stored by structure, e.g. the edges between the node and the root and do not

need to be stored in the nodes themselves. This already provides a simple but effective way of reducing the number of contexts stored compared to a transition probability matrix of a full order model. In the matrix, every state is stored separately with every single context contained. In a tree, the contexts on the edges are shared by all nodes in the subtree.

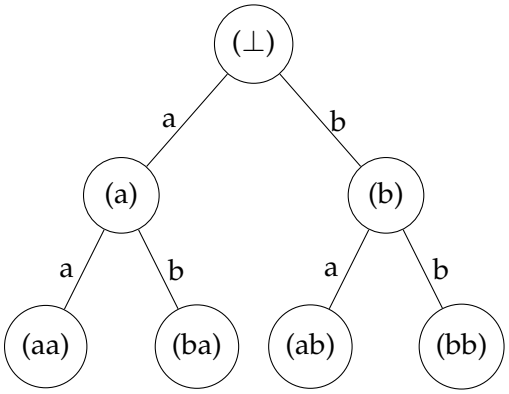

Figure 5.1: Suffix tree denoted with corresponding states

In the prediction suffix tree proposed by Ron et al., each node contains a pair $(s', p(\sigma|s = s'))$, where s' denotes the state represented by the node and $(s', p(\sigma|s = s')) \rightarrow [0, 1]$ is a function returning the transition probability for each symbol σ for the specific state s. The function $(s', p(\sigma|s = s'))$ can be determined in the same way as the $(s, p(\sigma|s))$ that are stored in the transition probability matrix in the Markov model as they simply represent the probability distribution of a single row in the matrix $\mathbf{P}'$.

In our application, it is sufficient to store the predicted context $succ(s)$ in the node for state s once the prediction was obtained using the transition probability function. An example of such a prediction suffix tree is shown in figure 5.2. The prediction for a state $s = \sigma_1 \cdots \sigma_k$ is stored in the deepest node that can be reached by a walk from the root along the edges $\sigma_k \cdots \sigma_1$. For example, the deepest node reachable for state abb is the node for state bb which stores a predicted context of b. Therefore, the prediction function for a prediction suffix tree is defined as

$$f(s) = \text{prediction stored in the deepest node that can be reached along edges } s$$

The prediction function represented by the tree in figure 5.2 is equal to a full order Markov prediction model with transition probabilities stored for all states of order less than or equal to a maximum order n.

5.3 Exploiting the Variable Order Property

The goal of lossless compression is to remove states and predictions under the constraint that the prediction function f must not change. However, by removing the node for state $s = \sigma_1 \cdots \sigma_k$ in a prediction suffix tree, the prediction falls back to the prediction stored for

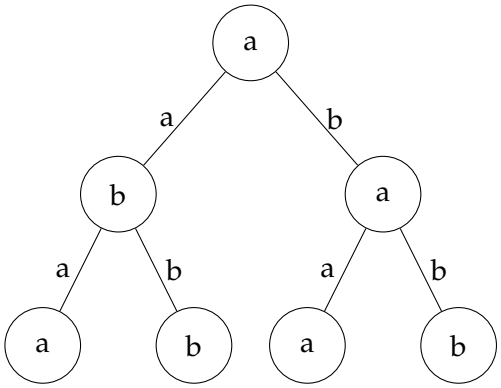

Figure 5.2: Suffix tree containing state predictions

state $suffix(s) = \sigma_2 \cdots \sigma_k$, which is possibly different. Therefore, we may only remove states s from a suffix tree for which the following condition holds:

$$f(s) \equiv f(suffix(s))$$

In the example tree, this is the case for instance for state ba. The predicted context for ba is b. If the node for ba was removed, the deepest node reachable would be the node for the suffix state a, which contains the same prediction b. Therefore, the node for ba can safely be removed without changing the prediction function. This also applies to the node for state b which predicts a just as the root node. However, this case needs to be handled differently because b has children nodes and can therefore not be simply removed. So instead of removing the node, we only remove the prediction of the node. When looking for predictions and encountering such an empty node, we simply proceed as if this node would not exist and return the prediction stored in the parent node. This makes it possible to also remove the node for state ab, which falls back to the root node as well. The prediction function of the resulting prediction suffix tree in figure 5.3 is equivalent to the prediction function given by the uncompressed tree in figure 5.2. In this example we were able to remove 4 nodes completely and could remove one prediction from a node. By the compression, the number of contexts in the tree could be reduced from 13 to 8.

5.4 Learning

Since the size of the uncompressed tree can become very large, it is wise to incorporate the compression algorithm into the algorithm building the tree rather than compressing a tree already built. The learning algorithm given in algorithm 2 will generate a prediction model equivalent to the classic Markov predictor. However, the number of predictions stored is reduced since some states are omitted during the building process. Next to the training sequence $\mathcal{T}$, the algorithm requires a maximum order n as input similiar to the full order Markov model.

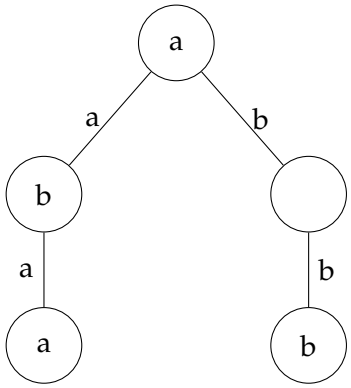

Figure 5.3: Compressed suffix tree

Algorithm 2: Learning of a lossless prediction suffix tree (PST)

```
1 Let T be a tree with only a root node
2 foreach k = 1...n do
3 |   S_k = {s ∈ S_k | s ∈ T}
4 |   foreach s ∈ S_k do
5 |   |   if f(s) ≠ succ(s) then
6 |   |   |   add prediction node corresponding to s to T, if necessary add missing
  |   |   |   empty nodes
7 |   |   end
8 |   end
9 end
```

The algorithm starts with only the prediction for the root node. It then examines states of increasing order and evaluates if a corresponding node must be added to the tree. A node is added only if the node's prediction $succ(s)$ differs from the prediction $f(s)$ already stored in the tree for some suffix of s. When a node for state s is added, the algorithm may need to add missing nodes between the node for the existing prediction $f(s)$ and the new node. This can happen if the parent node of s was previously omitted. No prediction has to be stored in these intermediate nodes since as they were omitted, their prediction is already correctly stored in the parent node.

To ensure correctness, it is essential to only start looking at nodes of order k after all nodes for order $k-1$ have been examined, e.g. have either been added or omitted. Otherwise, the fallback prediction of a higher order node that was omitted may be changed by the later addition of a suffix node. This is illustrated in figure 5.4: The root node and the node for state aa both predict a, and the node for state a predicts b. Let us assume that aa is considered for addition before a. The tree, which then only consists of the root node, predicts aa correct. Therefore it is not added. However, when a is added in the next step, the deepest node reachable for state aa is no longer the empty state but the node for state a, which contains an incorrect prediction for aa.

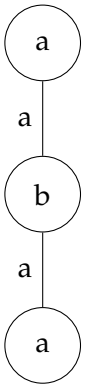

Figure 5.4: Omitting nodes

Apart from reducing the order, the size of the tree produced by the lossless compression cannot be influenced. However, some states of higher order may be more important to the prediction than states of lower order. For example, a state of higher order may be more accurate in predicting the future context because it has more information about past contexts available. In the next chapter, we will change the sequence in which states are added according to their importance measured by a rating function.

6 Lossy Compression

The lossless compression does not allow for specifying how much the model needs to be compressed since it simply removes the unnecessary states. Usually, there is only limited space available for the storage of a prediction model. Given a limit for the number of predictions stored, the goal is to reduce the predictions stored accordingly. In case the lossless compression algorithm does not remove enough states from the uncompressed model, a lossy compression mechanism needs to be employed. This however reduces the prediction performance by removing states for which the fallback prediction is different from the actual calculated prediction. The prime goal of lossy compression is to choose those states whose impact on the prediction performance is minimal. To this end, we propose a metric for rating each state accordingly.

The algorithm for learning the prediction suffix tree as introduced in [RST96] (algorithm 3) can be seen as a lossy compression algorithm. It uses a set S which contains all states to be added to the tree. The parameter $\varepsilon \in [0, 1]$ controls what states are added to the tree:

- Only states s are added whose transition probability function $p(\sigma|s)$ significantly differs from that of its parent describing the suffix of s. If they are roughly equal, they reason that the underlying process is not of order $|s|$ because it can be sufficiently described by the suffix of s. To this end, an error measure $Err(s, suffix(s))$ is introduced to compare the probability distributions of s and its suffix. Only if $Err(s, suffix(s)) \geq \varepsilon$, a node for state s gets added to the tree. Err is computed by using the Kullback Leibler divergence [KL51] of the two probability functions assigned to the states.
- Furthermore, contrary to the lossless mechanism where all states where added, they only consider states whose probability $\hat{p}(s)$ in the sample set exceeds ε.

Thus, the parameter ε controls how large the tree will grow. The higher ε, the less states will meet the constraints. However, there are some problems with it in our application:

Firstly, Ron et al. do not propose a way to determine ε. For their examples, values of 0.001 and 0.0001 are being used without further explanation of how these values were chosen. The size of the tree is difficult to control by choosing ε since its effect on the tree size is not known until the tree has been built. By trying different values, one can determine the ε which reduces the tree as desired. However, this is a costly process since it requires executing the algorithm numerous times. Furthermore, the Kullback Leibler divergence compares whole probability distributions. This is necessary if these distributions are being used later. However, we are not interested in the distributions, but only in the most likely successor. That means, even if two distributions are significantly different, we do not need to add nodes for both to the tree unless the prediction is different.

Algorithm 3: Learning of a PST as proposed in [RST96]

```
1  Let T be a tree with only a root node
2  Let set S = {σ | σ ∈ Σ ∧ p̃(σ) ≥ ε}
3  while S is not empty do
4  |  remove any s from S
5  |  if Err(s, suffix(s)) ≥ ε then
6  |  |  add a node v corresponding to s to T
7  |  end
8  |  if |s| < n then
9  |  |  foreach σ ∈ Σ do
10 |  |  |  if p̃(σs) ≥ ε then
11 |  |  |  |  add σs to S
12 |  |  |  end
13 |  |  end
14 |  end
15 end
```

Therefore, this algorithm is not suitable for our application. We will introduce a modified algorithm instead which adds the states in a predefined order - the rating - until the tree size reaches the given limit.

6.1 Ratings

Due to the limit of predictions stored, we should add those states' predictions first that improve the predictors performance most. Hence, a rating $g(s)$ should be based on properties which have an impact on the general performance of the predictor. The more positive this impact is, the higher the numerical value of $g(s)$ should be. We will use the following properties and calculate them based on the training sequence:

- $fre(s)$: How frequently is the prediction required, e.g. how high is the probility of encountering the state? The more often a state occurs, the more important it is that we have a prediction in the tree.
- $acc(s)$: How often does the prediction succeed in predicting the correct context? States with predictions that are correct with high probability have a more positive impact than states whose successor context is difficult to predict.
- $diff(s)$: How much does the prediction improve the accuracy compared to the currently stored prediction? Since we gradually build the prediction suffix tree, for any state s there is already a fallback prediction stored in the tree. The larger the difference between the currently stored prediction and the new prediction is for a state s, the higher we rate the state and its prediction.

We define the frequency measure of a state s as the probability that we observe state s when picking a random position in the training sequence. The training sequence contains $|\mathcal{T}|$ elements and the number of times a state s occurs in the training sequence is given by $c(s)$. We can therefore determine how frequent a state is by

$$fre : \mathcal{S}_{\leq n,\mathcal{C}} \to [0,1] : s \mapsto \frac{c(s)}{|\mathcal{T}|}$$

For the accuracy measure, we use the empirical probability $acc(s)$ that the prediction $f(s)$ was correct in the training sequence. We adjust these probabilities using LaPlace's rule of succession to the number of different contexts seen in total in the training sequence. Let σ_{max} be the context predicted for s. Then the accuracy is defined as

$$acc : \mathcal{S}_{\leq n,\mathcal{C}} \to]0,1] : s \mapsto \frac{c(s\sigma_{max}) + 1}{\sum_{\sigma \in \mathcal{C}} c(s\sigma) + |\mathcal{C}|}$$

However, when adding a prediction, we have to account for that in prediction suffix trees, there is a fallback prediction stored for every state. For example, let us assume a state s for which the prediction σ_a is stored in the tree. Let s' denote a possible child node for s, hence the fallback of s' is $suffix(s') = s$. Further, we assume that s' was followed either by σ_a or σ_b in the training set.

The prediction for s' is the most likely context $succ(s') = \text{argmax}_\sigma\ c(s'\sigma)$. Without loss of generality, let $succ(s') = \sigma_b$. Then the maximum number of correct predictions in the training set is $c(s'\sigma_b)$. If s' is not added, the prediction falls back to $f(suffix(s')) = \sigma_a$. The number of correct predictions in this case is $c(s'\sigma_a)$, that is the number of times s' was followed by the fallback context σ_a in the training sequence. The closer the difference between these numbers, the less the prediction improves by adding s' as a child to s.

This difference between both predictions is accounted for by the difference metric:

$$diff : \mathcal{S}_{\leq n,\mathcal{C}} \to [0,1] : s' \mapsto 1 - \frac{\text{correct predictions if } s' \text{ is omitted}}{\text{correct predictions if } s \text{ is added}} = 1 - \frac{c(s'f(suffix(s')))}{c(s'succ(s'))}$$

For $diff(s') = 1$, no correct predictions are made in case s' is not added, therefore the difference is maximal. On the other hand, for $diff(s')$ close to 0, the difference between adding and omitting s' is marginal with respect to the number of correct predictions in the training sequence. For $diff('s) = 0$, the fallback prediction is equally good. This metric replaces the Kullback Leibner divergence used by Ron et al. [RST96] before adding new nodes in the algorithm. Our metric is better suited for our application because it only compares predictions. Furthermore, unlike the Kullback Leibner divergence, it is straightforward to interpret: A value of $diff(s') = 0.6$ means that omitting the prediction for the state s' reduces the number of correct predictions for s' by 60% compared to the number of correct predictions if it is included in the tree.

All three metrics influence the predictor performance: A prediction with a good accuracy that is only rarely used is around as useful as a rather inaccurate prediction that can be applied

	s	$suffix(s)$
times succeeded by a:	12	13
times succeeded by b:	11	11
times succeeded by c:	10	12
times succeeded by d:	2	45

Table 6.1: low $acc(s)$, high $diff(s)$

often. The accuracy and difference measures may seem similiar at first, but they are both required. For a low $diff$ rating of state s, the accuracy is low too as the fallback prediction must have a high probability for occuring after s. This is of course unless $diff(s) = 0$, in which case the accuracy may still be high but since the fallback prediction matches the new prediction, no new node must be added regardless the accuracy. But the reverse does not hold, as illustrated in the example given in table 6.1: The accuracy of node for state s is $\frac{12+1}{12+11+10+2+4} = 0.33$ and thus only slightly better than choosing a successor randomly. However, adding a as the prediction safely results in 12 correct predictions in the training set. If it is not added, we only get 2 correct predictions since the fallback prediction is d. The difference measure in this example is $diff(s) = 1 - \frac{2}{12} = 0.83$ and indicates that the number of correct predictions falls by 83% when using the fallback prediction instead of the calculated prediction.

Hence, all three properties have a positive impact on the prediction. Consequently, our proposed rating is

$$g_1(s) = acc(s) * fre(s) * diff(s)$$

The difference rating needs to be recalculated constantly as the tree changes because it depends on the predictions stored in the tree. The accuracy and frequency ratings only have to be determined once for every state and are therefore easier to compute. While we expect the rating $g_1(s)$ to perform better with the difference measure included, it is interesting to examine how much this rather costly computation gains in terms of accuracy in the evaluation compared to the rating g_2 defined as

$$g_2(s) = acc(s) * fre(s)$$

Furthermore, as a benchmark to these ratings we define the rating g_3, which rates higher ordered states lower:

$$g_3(s) = 1 - \frac{|s|}{n}$$

So when using g_3, states are simply added in ascending order to the prediction model. In the next section, we will describe a modified tree generating algorithm which uses a rating $g(s)$ to construct the tree.

6.2 Prediction Suffix Tree with Rating

Algorithm 4 incorporates the rating into an algorithm that constructs the prediction suffix tree. The algorithm first adds all states from the training sequence up to a maximum order to the set S. For high orders, this set can quickly become pretty large. Therefore, we only include states with a minimum probability ν, where ν should be chosen sufficiently small. Then the states are gradually added to the tree starting with the state having the highest rating. We only add states whose prediction accuracy would decrease by more than ε if omitted. The purpose of ε is not to compress the tree, but to keep the algorithm from adding predictions in case the existing tree already provides an almost equivalent prediction.

Algorithm 4: Learning of a PST with Rating

```
1  Let T be a tree with only a root node
2  Let set S = {s ∈ S_{≤n} | s ∈ T ∧ fre(s) ≥ ν}
3  Let set S_o = {}
4  while S is not empty ∧ size left do
5  |  remove any s from S so that ∀ s' ∈ S : g(s) ≥ g(s')
6  |  if diff(s) ≥ ε then
7  |  |  add fallback nodes if suffix(s) not stored in tree
8  |  |  add prediction node for s to T
9  |  |  foreach s' ∈ S_o in ascending order do
10 |  |  |  if s ∈ suffix*(s') ∧ diff(s') ≥ ε then
11 |  |  |  |  add fallback nodes if suffix(s') not stored in tree
12 |  |  |  |  add prediction node for s' to T
13 |  |  |  end
14 |  |  end
15 |  else
16 |  |  add s to S_o
17 |  end
18 end
```

The states that do not match this critera can however not simply be discarded because $diff(s)$ changes as the tree changes in certain situations: It is possible that a state s has a better rating than one of its suffix states $s' \in suffix^*(s)$. In this case, the prediction for s is considered for inclusion prior to s'. Let us assume that s' is not added because the fallback prediction is sufficient and the prediction for s' differs from the prediction for s. Then the inclusion of s' can change the difference rating of s. A similiar problem made it necessary to add states in ascending order in the variable order model on page 27 because otherwise the prediction may change with the addition of a new node. However we can not avoid this problem in the same way as the states are supposed to be added according to their rating. Since usually not all states are being added because of the size limit, changing the sequence of states added will change the outcome of the algorithm. Therefore, after adding a state s', we must check if the difference rating changed for states that were omitted and would have been added to

the subtree of state s'. For states whose difference rating exceeded ε, a new node must be added.

6.3 Prediction Suffix Tree with Rating and Lookahead

Algorithm 4 add up all states up to the maximum to the set S. This is a problem because the number of possible states generally grows exponentially with the order. Therefore, for large training sets it becomes increasingly elaborate to find and create ratings for all states. The execution of the algorithm can quickly become infeasible on mobile devices as they only have limited computational power. Algorithm 3 avoids this problem by successively adding states to the set S after their suffix state was added to the tree. There is a catch to the gradual adding of states to S though. The algorithm does not compute the rating of state s before adding its suffix state s'. That means that high ranked states are only looked at if their possibly lower ranked suffix state has been added to the tree. We can cope with this problem to some extend by increasing the scope of S. To this end, when removing a state s from the set S, instead of adding only its children to S, we add all states in its lookahead set $la_\alpha(s)$:

$$la_\alpha(s) = \{s' : s \in suffix^*(s') \wedge |s'| \leq (|s| + \alpha) \wedge fre(s) \geq \nu\}$$

$la_\alpha(s)$ contains all nodes in the subtree of the node for state s that can be reached by a walk on at most α edges from s, e.g. have an order of at most $|s| + \alpha$. The parameter α determines the size of the set. For example for $\alpha = 1$, only direct children of s are added. Algorithm 5 uses this lookahead by gradually adding new states to the set S. This gradual adding is in particular beneficial when the size limit is small but there are many possible states. In this case, the original algorithm first needs to add and determine ratings for all of these states, even if the algorithm stops after a few iterations because the size limit was reached. With a lookahead set, fewer states are added initially which speeds up the calculation of the state to add next in each iteration.

6.4 Continuous Learning of the Prediction Suffix Tree

The continuous predictor learns the transition probabilities gradually. Here, a different compression approach is required since there is no training sequence available that we can base predictions or a state rating on. In the previous chapter, we already introduced a method to construct a prediction suffix tree starting from the root by adding new 'good' nodes. We cannot use this algorithm, but we can modify it to handle the initial lack of knowledge. Instead of storing the predictions itself in the nodes, we store the information required to calculate the prediction, that is the probability distribution for the successor contexts. This makes it possible to update the predictions over time and gradually expand the tree as the information is updated with every transition. Due to the size limit, we cannot store the probability distribution for all states of any order seen. Without the probability distribution however, we cannot determine what states should be stored because no rating

Algorithm 5: Learning of a PST with Rating and Lookahead

```
1  Let T be a tree with only a root node
2  Let set S = {s ∈ S_1 | s ∈ T ∧ fre(s) ≥ ν}
3  Let set S_o = {}
4  while S is not empty∧ size left do
5  |  remove any s from S so that ∀ s' ∈ S : g(s) ≥ g(s')
6  |  if diff(s) ≥ ε then
7  |  |  add fallback nodes if suffix(s) not stored in tree
8  |  |  add prediction node for s to T
9  |  |  S=S ∪ la_α(s)
10 |  |  foreach s' ∈ S_o in ascending order do
11 |  |  |  if s ∈ suffix*(s') ∧ diff(s') ≥ ε then
12 |  |  |  |  add fallback nodes if suffix(s') not stored in tree
13 |  |  |  |  add prediction node for s' to T
14 |  |  |  end
15 |  |  end
16 |  else
17 |  |  add s to S_o
18 |  end
19 end
```

can be computed. Hence, we have to take a different approach here. Our algorithm expands the tree at nodes for which we expect the maximum improvement in prediction accuracy by increasing the order. Since nodes that already have a good accuracy are more unlikely to improve than nodes with low accuracy, we are only considering nodes with an accuracy smaller than a desired minimum accuracy μ:

1. Identify states in the tree with $acc(s) < \mu$
2. Try to improve the prediction by also observing context transitions for higher order children.
3. If the size limit is exceeded, remove states whose predictions do not differ from their parents ($diff(s) = 0$) or that have not been seen for a long time.

Algorithm 6 updates the count function $c(s\sigma)$ for all states s in the tree. The values of $c(s\sigma)$ are required to determine the transition probabilities of state s to contexts σ like in the uncompressable model on page 21.

The algorithm starts with a root node only. For every transition, the algorithm compares the prediction with the actual context. If they match, it only counts the transition for all corresponding nodes in the suffix tree by increasing the count function c. If the prediction in the tree is incorrect and the accuracy of the node is smaller than the desired accuracy μ, the node is expanded by adding a child according to the context transition observed. To prevent that new nodes expand too quickly, we require a minimum number β of occurences

$c(s')$ of the state s' before a child node may be created. When the size limit is exceeded, we remove leaf nodes whose difference rating is very low and - if no such nodes exist - leaf nodes used least recently. Note that during the algorithm, we do not add nodes for states where $diff(s) = 0$ holds. However, the nodes' predictions can change over time and such nodes could evolve. By the least recently seen metric we try to estimate the frequency rating which we cannot calculate otherwise. When removing a leaf node, we also clear the values of c that were required to calculate the transition probabilities for the node's state, thus freeing memory until the size has been reduced to the given limit.

Algorithm 6: Continous Learning of a Prediction Suffix Tree

```
1  Let T be a tree with only a root node
2  Let set S = {σ|σ ∈ Σ}
3  foreach transition s → σ do
4      if T(s) ≠ σ then
5          Let s' be the state storing the prediction for s
6          if acc(s') < μ ∧ |s'| < n ∧ c(s') ≥ β then
7              Choose σ' so that σ's' ∈ suffix*(s)
8              add child for context σ' to s'
9          end
10     end
11     foreach state s' ∈ S_T : s' ∈ suffix*(s) do
12         c(s'σ) = c(s'σ) + 1
13     end
14     while size exceeded do
15         Find leaf node v for state s with diff(s) < ε and remove v
16         If no such node exists, remove leaf node v for state s least recently seen
17         clear c(sσ) for all σ
18     end
19 end
```

7 Application in a Distributed Scenario

So far we introduced various methods to store and compress prediction models. In the distributed scenario described in chapter 3 it is necessary to store several prediction models on the consumer device which has limited memory dedicated to storing those models. This necessitates a mechanism to compress a set of models.

We assume that the number of contexts the consumer is able to store is given by L and a set of m prediction models $M = \{M_1, ..., M_m\}$ needs to be stored. The goal of compressing multiple models is to compress them in a way so that their total size does not exceed L at a minimal impact on the prediction accuracy. Therefore, for each model M_i we need to determine a size l_i so that $\sum_i l_i \leq L$.

Unlike pre-trained models, continuous variable order models change permanently because they learn the transition probabilities constantly. Therefore, we need to differentiate between trained and untrained variable order models.

7.1 Compression of Multiple Trained Variable Order Models

The lossy algorithms introduced were compressing a trained model during its generation. Based on the information obtained from the training sequence, they added predictions in a certain order until no more states were left or the size limit was exceeded. This was done to avoid the computational effort required for building larger models than actually allowed and compressing it again.

However, in cases we do not know the exact space allowed, we may need to compress an already trained model further. To this end, we can simply invert the order in which predictions were added to the model during its generation, thus removing the prediction first that was added last to the model. Because we added predictions for states in descending order of their rating $g(s)$, the state last added must have the lowest rating among all other predictions in the model. Because the rating is equally defined for all models M_i, it can also be used to compare predictions between different models. Our approach (algorithm 7) is therefore to find the model M_i with the lowest rated prediction, remove this prediction and repeat this process until the total size dropped to the given size limit L. To this end, we maintain an array Min which is initialized with the lowest prediction rating of each predictor. Then the prediction model M_r with the lowest rating is determined and the corresponding prediction removed. Since the worst rating of M_r now changed because we removed the

minimum prediction, we must update the corresponding value stored in Min. These steps are repeated until the predictors have been sufficiently compressed.

Algorithm 7: Compression of a set of trained variable order models

```
1 Let M = {M_1, ..., M_m} be a set of prediction models with initial size l_i = L
2 foreach M_i ∈ M do
3 |   Min[i]=g(s) : g(s) ≤ g(s') ∀ s' ∈ M_i
4 end
5 while Σ l_i ≤ L do
6 |   r = r : Min[r] ≤ Min[k] ∀ k = 1..m
7 |   remove lowest rated prediction from M_r
8 |   Min[r]=g(s) : g(s) ≤ g(s') ∀ s' ∈ M_r
9 end
```

The idea behind this algorithm is not to compress the predictors so that they perform equally well. Some predictors can inherently perform better than others, for example because the randomness in the data to predict varies. In such a case, trying to improve a poorly performing predictor by allocating more space to it is futile and not only leaves this predictor's accuracy unchanged but also significantly decreases the accuracy of another predictor that could have benefited from additional predictions. Therefore, we took the reverse approach by adding states to predictors that do well and removing states from predictors that don't perform well.

7.2 Compression of Multiple Continuous Variable Order Models

A continuous variable order model always starts with size 0 because it stores no predictions initially. Therefore, the compression algorithm has to be executed repeatedly (algorithm 8). Without having any information to begin with, the maximum allowed size l_i for each predictor is set to the average size. To distinguish between the allowed size and the size actually used, we denote the actual size of a prediction model M_i by $|M_i|$. This allocated size is only changed when it is necessary, that means when a prediction model M_i has less than ω size left, e.g. $l_i - |M_i| \leq \omega$. The allocated space l_i for this predictor is then increased by ω, if several such predictors exist we choose the one with the least space remaining. By increasing an l_i however, the sum $\sum l_i$ exceeds the size limit L. Since the predictors usually increase in size over time, at some point also the total space used $\sum |M_i|$ will have grown beyond L. When this happens, we choose a predictor to compress and set the allocated size one less than the size it currently uses. We designed the continuous variable order model so that it can deal with that reduction itself by removing an unnecessary or rarely used prediction.

Algorithm 8: Compression of a set of continuous variable order models

```
1  Let M = {M_1, ..., M_m} be a set of prediction models with initial allowed size l_i = L/m
2  foreach Context Transition do
3  |  find i : l_i − |M_i| ≤ ω ∧ ∀ j : l_j − |M_j| ≥ l_i − |M_i|
4  |  if such i exists then
5  |  |  l_i = l_i + ω
6  |  end
7  |  while Σ_i |M_i| > L do
8  |  |  choose predictor M_i to compress
9  |  |  l_i = |M_i| − 1
10 |  end
11 end
```

The question remains however how to define the function chosing a predictor to compress in line 8. We propose two ways, a size based selection and a time based selection mechanism:

- The size based mechanism chooses the largest model to compress, thus $M_i = \text{argmax}_{M_j} |M_j|$
- The time based mechanism chooses the model storing the least recently seen state. Let s be a state for which M_j stores a prediction for. Then $t(M_j, s)$ denotes the number of transitions that passed since the prediction for s was needed last. We define $t(M_j)$ as the maximum of $t(M_j, s)$ over all states stored in M_j: $t(M_j) = \text{argmax}_{s \in M_j} t(M_j, s)$. The predictor from which a prediction is to be removed can now be obtained by again taking the maximum value of these $t(M_j)$ for all models: $M_i = \text{argmin}_{M_j} t(M_j)$

Both approaches have advantages and disadvantages. In cases the size restriction is strong and all prediction models require more than average size, the space allocated by the based approach will oscillate around $\frac{L}{m}$ because it always compresses the largest model. It is therefore better suitable for larger L. The performance of the time based approach depends on how well the least recently seen metric is suited to the data. For instance, a useful and generally often needed prediction may be removed only because it has not been seen recently. This however is a general problem of the continuous variable order predictor which is unable to determine state probabilities because it only keeps track of a changing subset of states.

8 Evaluation

We conducted an evaluation on real world movement traces and generated data. We will start with examining the trained and untrained predictors individually. Then a separate evaluation will show how the predictors perform when being conjunctly compressed in the distributed scenario that motivated the compression. Eventually, we will repeat some of the evaluations on generated higher order traces in section 8.6.

Predictor	Type	Parameters
Fixed Order	trained	
Full Order	trained	
Continuous Markov	untrained	
Variable Order lossless	trained	
Variable Order lossy compressed	trained	min. difference $\varepsilon = 0.01$ min. probability $\nu = 0.0005$ lookahead $\alpha = \infty$
Continuous Variable Order	untrained	min. difference $\varepsilon = 0.01$ min. transitions $\beta = 4$ accuracy $\mu = 0.75$

Table 8.1: Predictors and default Predictor Parameters

Table 8.1 provides an overview of the predictors and their default parameters used in the evaluation. For the lossy compressed variable order predictor we chose $\varepsilon = 0.01$. Evaluations showed no difference in accuracy for values lower than 0.01 and only a very minimal decline for $\varepsilon = 0.1$. The value of ν was chosen very low so it does not affect the performance but at the same time provides a small boost in speed by discarding highly unlikely states with a probability of less than 0.05%. As far as the lookahead α is concerned, we used an infinite lookahead by default but also evaluated how values of $\alpha = 1$ and $\alpha = 2$ influenced the prediction result.

8.1 Real World Data

We evaluated the prediction models using movement traces based on access point connectivity that were collected on the campus of Darthmouth College between 2001 and 2004 [CRA]. At that time, the campus had around 500 wireless lan access points covering most of the campus area. For each MAC address connected to the network, a trace was generated

containing the timestamp and access point identifier for each transition to another access point. A special location 'off' was introduced for situations in which the connectivity was lost, for example because the device was out of range or switched off. For our evaluation, we defined a context as the identifier of the access point the user is connected to, or 'off' respectively.

In total, 13.888 traces were generated. However, some of these traces are comparably small because the device was off or the user remained stationary most of the time. In order to have sufficiently large traces for training and testing the predictors, we extracted those traces with more than 10.000 access point transitions. Some traces consisted only of transitions between very few access points located in the same building. Therefore we filtered the traces further to users who visited at least 15 different access points in both the training set and the evaluation set and conducted our evaluation on 200 of these traces. In the following evaluations, we used the first 10.000 transitions of each trace. The first 5000 transitions were provided as the training set to predictors that require training. For the continuous predictors we skipped these transitions. We then evaluated the predictors' performance on the following 5000 transitions with respect to their accuracy and size. The accuracy is simply the ratio between the number of correct predictions and the number of total predictions made. If the predictor does not store a prediction, the prediction is also counted as incorrect. The size is measured by the number of contexts stored in the model, e.g. contexts stored as predictions for states and contexts necessary to store the states themselves. In our sample data the contexts can be represented by a short string, but for complex context information, more space is required. The continuous predictors store a discrete function to count transitions for each state in order to obtain the transition probability function. Here we assume that the memory required is equal to the number of contexts for which the function is defined as greater than 0.

8.2 Uncompressed Markov Models

To get a rough idea about the scale of orders involved in the traces and how the predictors can deal with them, we calculated the optimal order with respect to accuracy per trace (figure 8.1). To this end, we evaluated each trace for different orders for each of the uncompressed prediction models. The comparison shows that the median optimal order is 2 for all models examined. Nevertheless, the models show a different behavior:

The fixed order model is strongly biased towards lower orders. This can be attributed to the lack of a fallback prediction. Therefore it can only make predictions for states exactly seen in the training sequence. With higher orders and hence a growing number of possible states, it becomes increasingly unlikely to observe all states in the training sequence that are observed in the evaluation. This optimal order for the fixed order predictor rarely exceeds 2. The full order model does not suffer from states not seen because in such cases it can use a prediction for a suffix state. Consequently, in our traces it could exploit the order considerably better than the fixed order model. In around 50% of the traces, the optimal order was greater than or equal to 3. The same applies to the continuous Markov model. The difference between

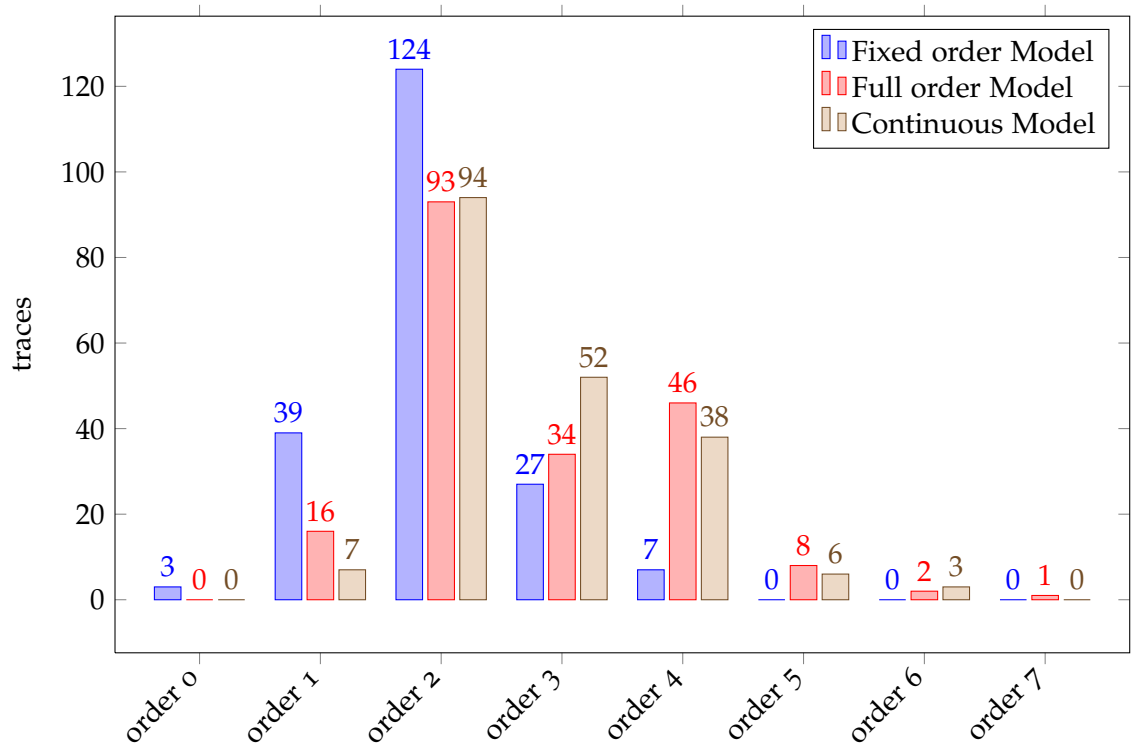

Figure 8.1: Optimal orders for individual traces

these two models can be attributed to the different learning methods employed. In the next section we will examine how the order impacts the accuracy and size of the models in detail.

8.2.1 Accuracy

First we examined the accuracy for various orders of the classical Markov model with fixed order, e.g. predictions can only be made for states of order n or higher. The cumulative distribution function in figure 8.2 shows the cumulated accuracy reached. The order 0 model equals the prediction of the most frequently seen context. The accuracy improves by increasing the order from 0 to 1 and from 1 to 2. At order 2, the maximum is reached with a median accuracy of 0.581. Increasing the order beyond 2 does not improve the prediction accuracy. On the contrary, the predictor of fixed order 3 is slightly worse than the order 2 predictor. For higher orders, the accuracy continues to fall at an increasing rate. This is likely to be caused by states in the evaluation sequence for which no prediction has been stored, e.g. states that did not occur in the training sequence. In that case, a prediction is counted as incorrect. This problem is far less substantial for lower order models. Nevertheless, an increased order not necessarily results in a reduced accuracy for orders greater than 2 as already indicated by the chart in figure 8.1.

Our assumption that the lack of fallback predictions causes this effect is backed by the result of the full order model (figure 8.3). Being able to make predictions also for lower orders, it performs best at order 4. The accuracy of the full order model is consistently better than the accuracy of the fixed order predictor. However, the gain in accuracy is rather small. The median accuracy increased by 0.028 to 0.609 compared to the best fixed order model. For higher orders the accuracy starts to slowly decrease again. Thus, even with fallback predictions, increasing the order does not necessarily improve the predictions. A reason could be that traces that are actually of lower order suffer from the overfitting that results from including too many states in the model.

The continuous Markov predictor shows a similiar behavior (figure 8.4). While by continuously updating the transition probabilities, the general accuracy could be improved significantly compared to the trained predictors, it fails to exploit traces that perform better at higher orders without suffering from a negative impact caused by inherently lower order traces.

In the next section, we will examine the cost in terms of memory usage for each of these predictors.

8.2.2 Size

We compared the number of contexts stored examplarily for the full order model in figure 8.5. The order 0 predictor only stores a single context to be predicted. As expected, the number of contexts grows rapidly as the order is increased because of the larger number of possible states. The predictor of order 4 which performed best with respect to accuracy has a median size of 4534 contexts. It is also notable that while most traces only differ moderately in size, there are a few outliers with a much larger number of contexts. For the other predictors, we observed a similiar behavior.

Figure 8.6 (page 48) shows a comparison for order 4 between the fixed order, full order and continuous Markov predictor model. Least space was required by the fixed order model which also stored the least number of states $\mathcal{S}_{4,\mathcal{C}}$. The full order Markov model of order 4 stores predictions for all states $s \in \mathcal{S}_{\leq 4,\mathcal{C}}$. The continuous Markov predictor only used marginally more space than the full order predictor. Nonetheless, the median improvement in prediction performance is 0.11 (see figures 8.2 and 8.4) and thus quite significant, in particular in relation to the additional space required. We have to keep in mind the additional cost induced by updating the transition probabilities though. For example, energy is consumed when transferring contexts to the subscriber and computing the new transition probability matrix.

In this section, we measured how the uncompressed predictors performed. We will use these results as a benchmark for the compressed models' performance in the next sections.

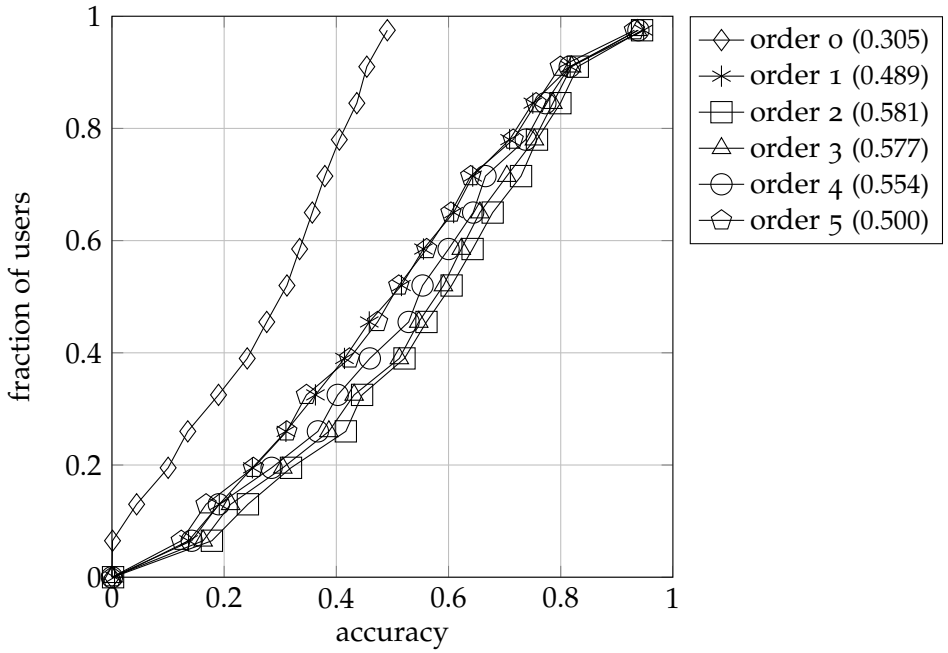

Figure 8.2: Accuracy for fixed order predictor ($n = 0 - 5$)

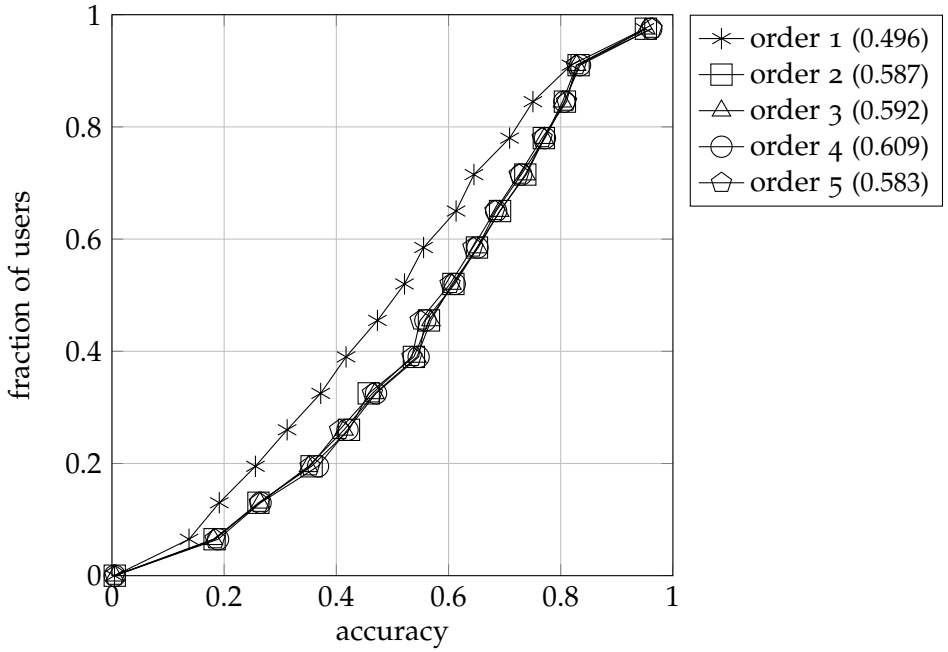

Figure 8.3: Accuracy for full order predictor ($n = 1 - 5$)

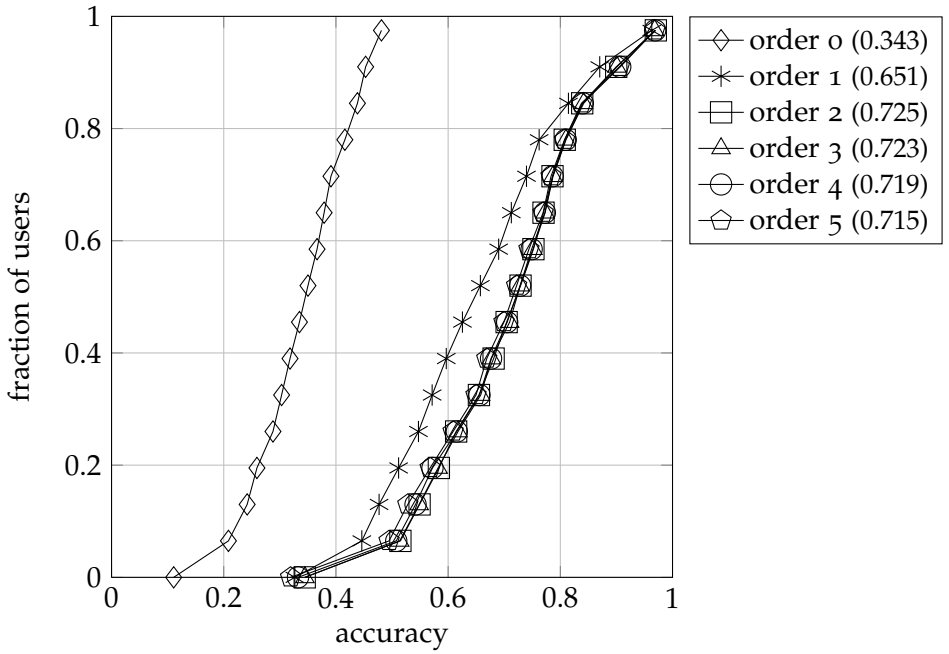

Figure 8.4: Accuracy for continuous Markov predictor ($n = 0 - 5$)

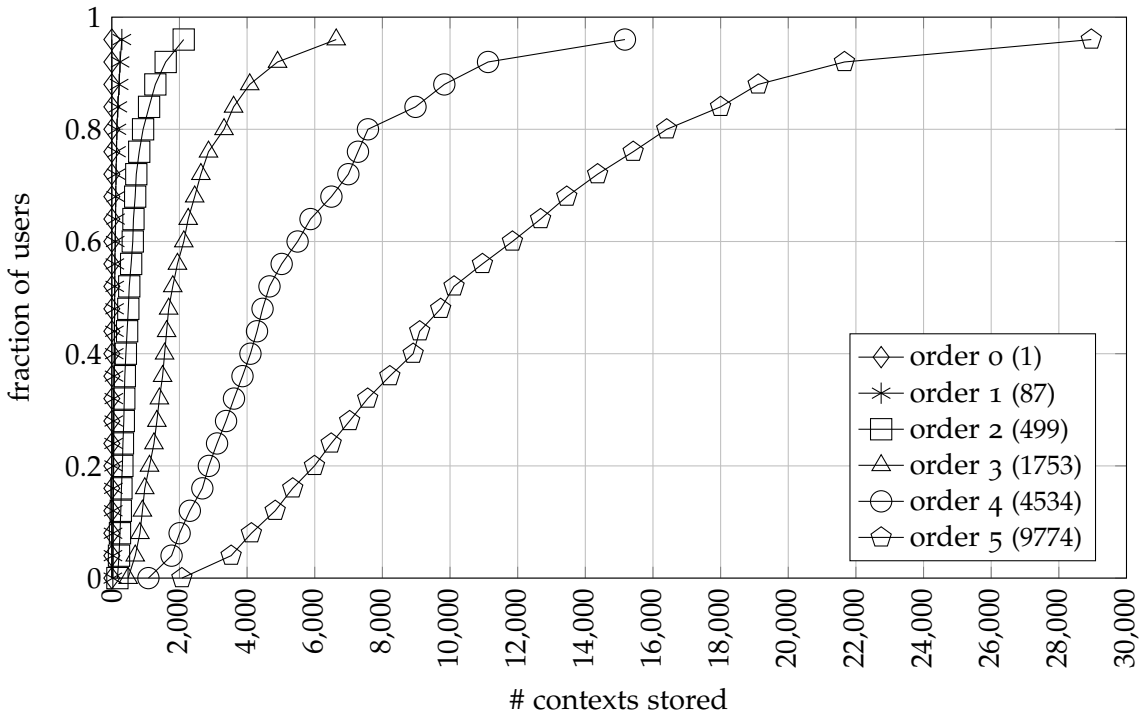

Figure 8.5: Stored contexts for full order predictor ($n = 1 - 5$)

8.3 Lossless Compression of the Full Order Model

The lossless compressed variable order model is equivalent to the full order Markov model with respect to prediction accuracy. However, it generally requires less space because it allows for omitting redundant states. Furthermore, space is saved by storing the predictions in a tree based structure instead of a matrix. We compared the size of the fixed order, full order and lossless compressed model exemplarily for order 4. The result is shown in figure 8.6. The lossless compressed variable order model is significantly smaller in size than the full order model and even than the corresponding fixed order model. The space required could be reduced to around 13% of the full order model.

The reduction compared to the fixed order model can be attributed to the elimination of some higher order states. Using the variable order model, several higher order states can be replaced by a single state of lower order, which moreover requires less contexts to be stored than a higher order state. To identify to what extend the lossless compression benefited from the structural compression and the compression by omitting predictions, in figure 8.7 we only counted the number of predictions stored in the model. That way, the storage mechanism (matrix or tree) is negligible and we can measure the effect of omitting predictions on the model. Apparently, in our setting the size of the models is predominantly determined not by the predictions themselves, rather than by the states the predictions correspond to. The predicted contexts only amount to around 25% (1029) of the total size (4534) in the full order model. Nevertheless, the variable order model managed to reduce this number by 74% to 269 without changing the prediction accuracy. We can therefore summarize that the lossless compression mechanism constitutes an efficient way to reduce the number of contexts stored for full order Markov models without any impact on accuracy.

8.4 Lossy Compression of Individual Models

8.4.1 Lossy Compression of the Variable Order Model

To measure the impact of compression on the accuracy of a variable order model, we first determined the lossless compressed model for each trace and then compressed the variable order model to a certain fraction of the original size using different ratings. The result is shown in figure 8.8 for a compression by 75%. The lossy compression performs remarkably well for all ratings. The median accuracy of the uncompressed model (0.609) is only missed by 0.001 by the rating g_1 which takes a state's probability, the prediction accuracy and the difference to the existing prediction into account. Removing the difference measure from this rating has almost no negative impact (0.001) on the result.

The result indicates that the prediction models contained many predictions which could not be removed by the lossless compression but whose removal had no or only a minor impact on the accuracy. Even the simple method of removing higher order states first managed to compress the models without much loss in accuracy. Compressing the models with a compression ratio of 75% seemed to easy. Therefore, we compressed the models to 90% and

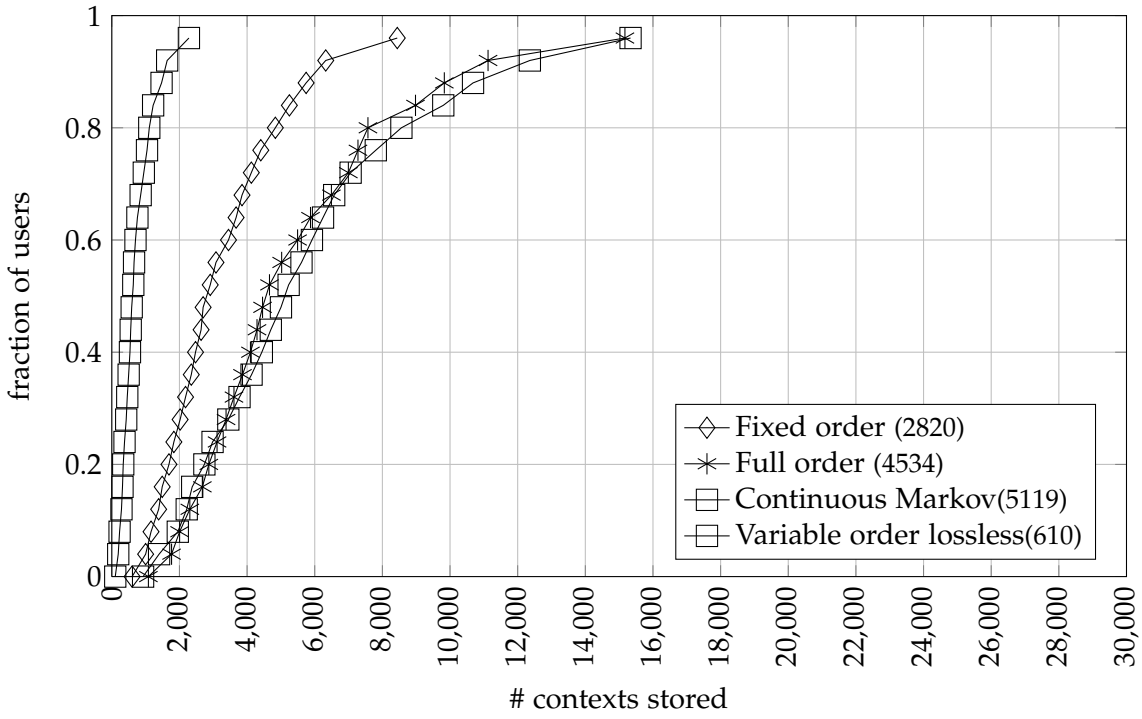

Figure 8.6: Stored contexts for order 4

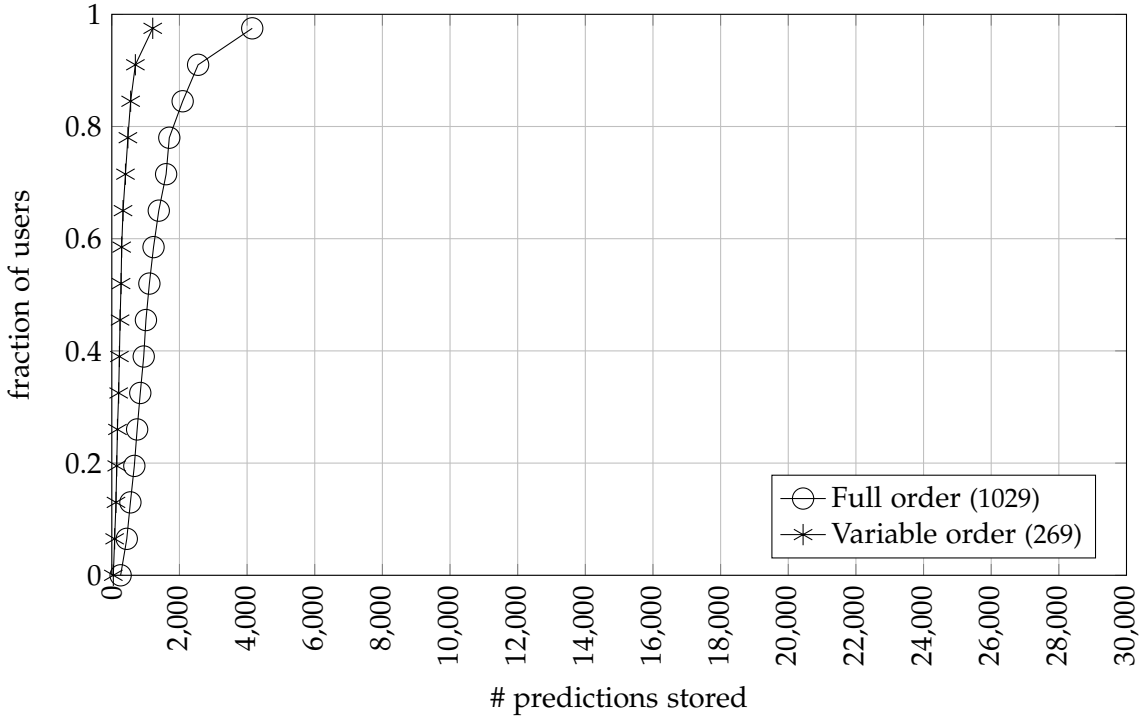

Figure 8.7: Stored predictions for order 4

99% to see how some of the ratings perform under tightened conditions (figures 8.9 and 8.9). The results show quite clear that the ordered based rating is inferior to the other two ratings examined. Furthermore, the effect of including the difference rating only becomes visible at very high compression ratios. We assume the reason is that with a tight size restriction, the majority of states to be added is of low order. For lower order states, the variance in predictions between a state and its suffix generally tends to be higher than for higher orders. For example, the empty state and a first order state are more likely to have completely different successor distribution functions than a state of order 4 and its suffix state. Therefore, the inclusion of the difference in the assessment of a state's rating is more important at lower orders and high compression ratios because the impact of a suboptimal decision is more severe.

Next, we examined how the size of the lookahead set effects the resulting predictor. The lookahead determines how many states the generation algorithm can compare for adding to the predictor. The larger α, the faster this set grows during the construction of the prediction model. To this end, we conducted the evaluation for $\alpha = 1$ and $\alpha = 2$ and compared the result to the accuracy of the original algorithm ($\alpha = \infty$). The result for the order 4 variable order predictor in figure 8.11 shows that the size of the lookahead has a noticeable effect on the accuracy. An increase in α from 1 to 2 results in a gain in accuracy of 0.024.

8.4.2 Lossy Compression of the Continuous Variable Order Model

The continuous Markov predictor constantly keeps track of the transition probabilities. Contrary to the continuous Markov predictor which stores all transition probabilities, the continuous variable order predictor does so only for selected states. In our traces, the continuous Markov predictor scored a median accuracy of 0.719. An average of 6118 contexts was stored in the model. We evaluated the impact on accuracy when compressing the model for each trace to 25%, 10% and 1% of the size of the corresponding uncompressed model. We discovered that the compression ratio only has a low impact on the prediction accuracy (figure 8.12). A compression to 25% reduced the median by 0.005. Even when compressing the continuous Markov predictor to 1% of its original size using our approach, the accuracy remains considerably high (0.656). We also compared this approach to the continuous predictor that makes predictions independently of the state. Such a predictor consists of the root node only and predicts the most likely context seen so far. This predictor has roughly half of the accuracy of the 1%-compressed predictor, however it stores only 11 contexts less.

8.5 Compression of Multiple Prediction Models

So far we examined how the predictors perform at different compression ratios. We motivated our thesis with the compression of multiple prediction models under a fixed time constraint. In this section, we will evaluate how the variable order and the continuous variable order model in such an environment. For this purpose, 25 traces were chosen,

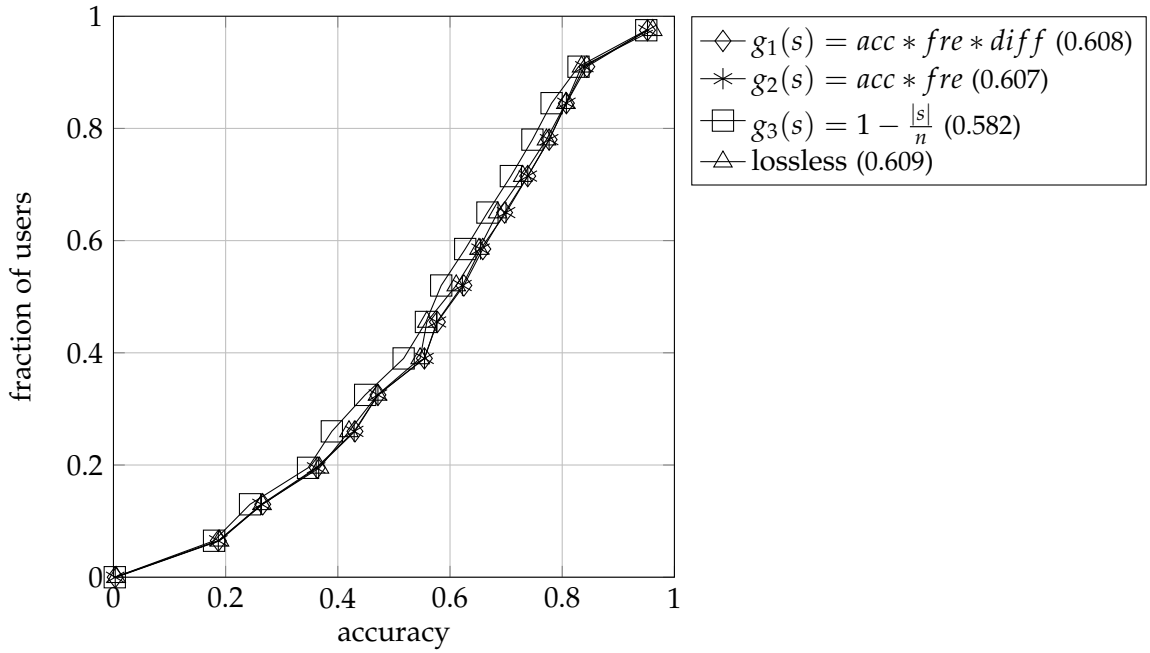

Figure 8.8: Accuracy for various ratings ($n = 4$ and 75% compression)

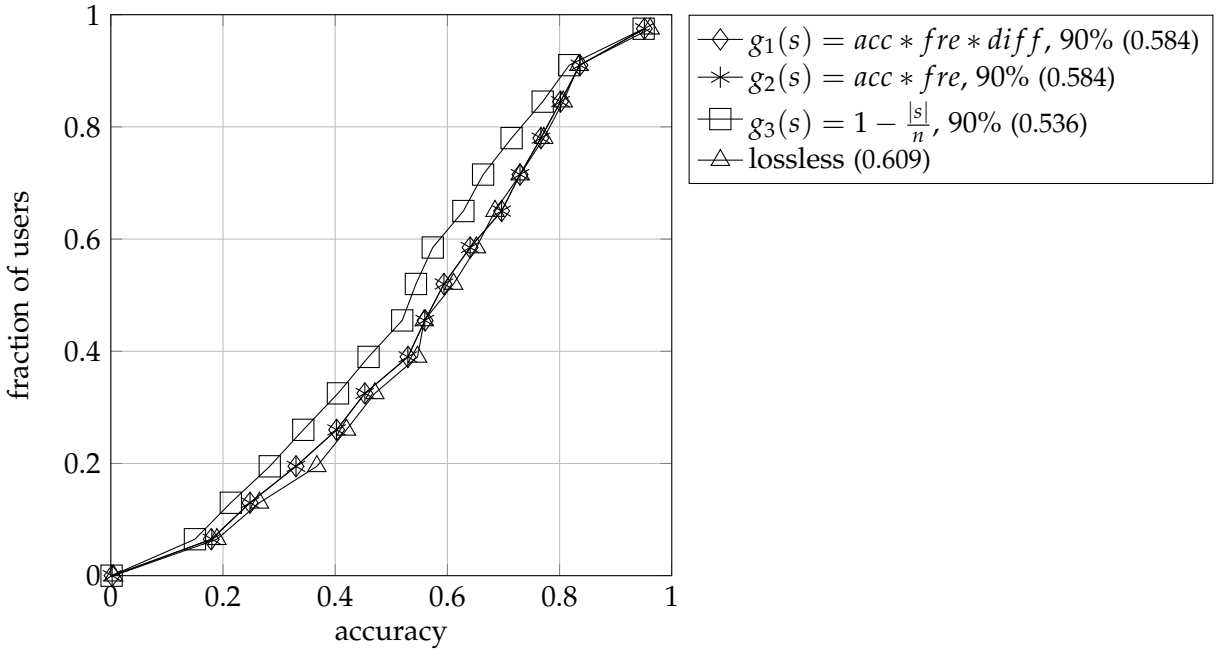

Figure 8.9: Accuracy for various ratings ($n = 4$ and 90% compression)

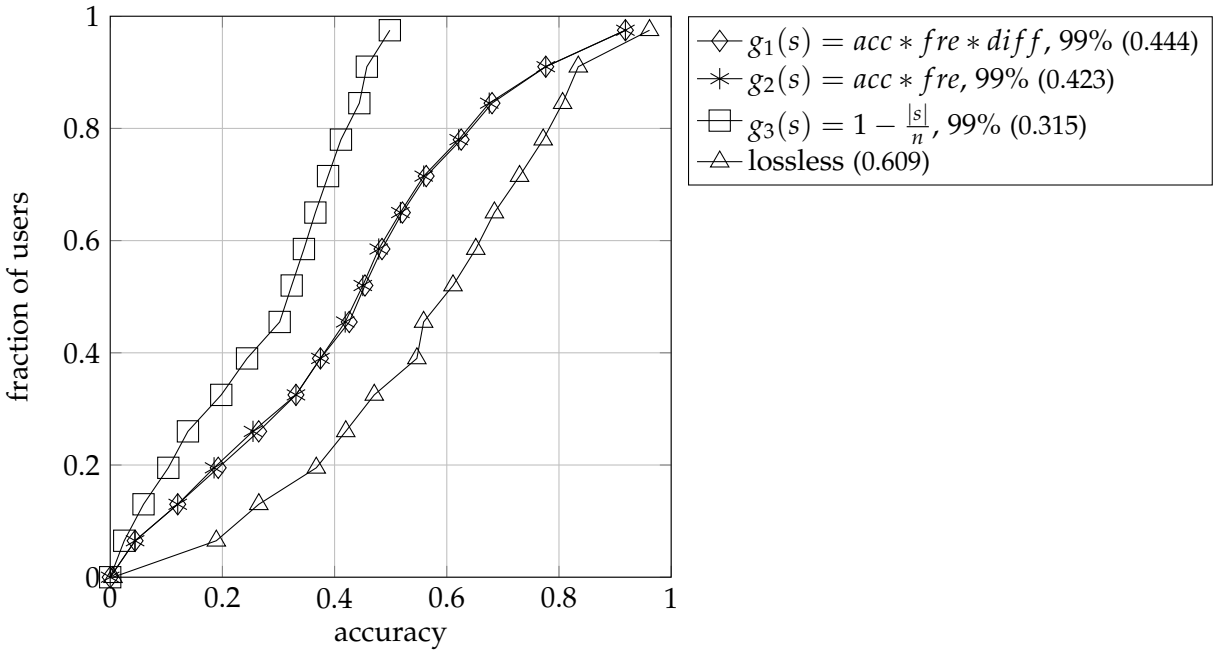

Figure 8.10: Accuracy for various ratings ($n = 4$ and 99% compression)

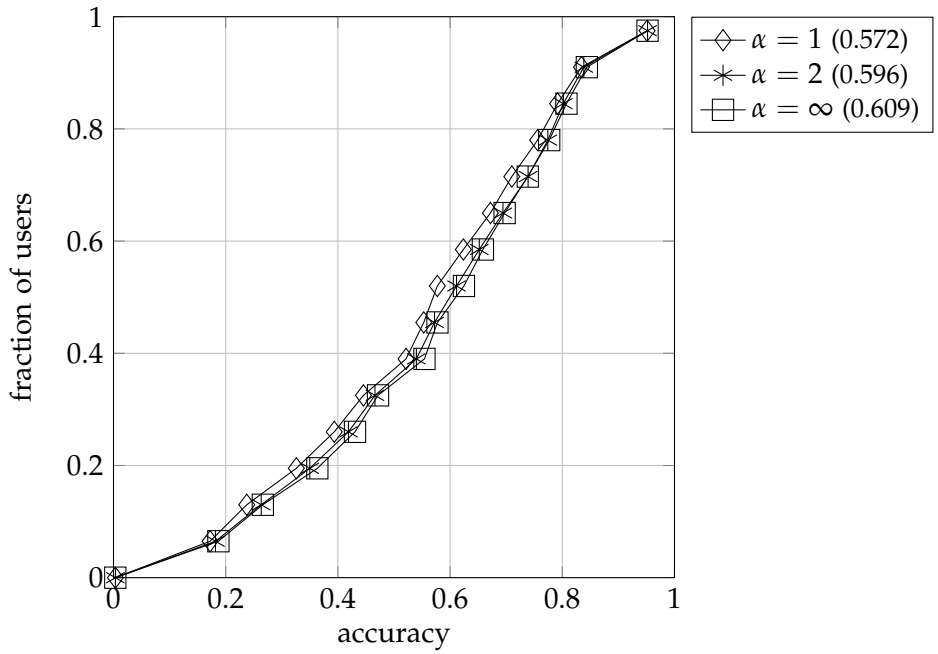

Figure 8.11: Accuracy for various α ($n = 4$, rating g_1 and 75% compression)

the lossy compression was applied for various size limits to the trained predictors and the resulting set of compressed predictors was evaluated on 5000 transitions.

8.5.1 Variable Order Predictor

We evaluated the performance of the variable order predictor for the rating $g_2(s) = acc * fre$ and $g_3(s) = 1 - \frac{|s|}{n}$. The results are shown in figures 8.13 and 8.14 respectively. The total size of the lossless compressed models was 17825 contexts with a median accuracy of 74,5%. As already indicated by the results in the previous section, when the predictors are compressed severly, the g_3-rating scores considerably worse than g_2. For $L = 125 - 1000$, g_3 consistently scored around 0.3 lower than g_2. This underlines the effectiveness of g_2-based compression which seems to be able to choose good states quite well. This is particularly important when only few predictions can be stored and choosing one prediction over another can have a significant impact on the accuracy. Even at a compression of 97,20% (L=500), the median accuracy deviated only by about 0.045 from the supposed optimum. Furthermore, for sizes greater or equal than 1000, the maximum and minimum accuracy reached is almost equal to the lossless compressed models. Unlike for the g_3 rating, we did not observe any outliers with extremely low accuracies, a property that can prove to be useful in applications. At a size limitation of 4000, our compression algorithm actually marginally outscored the lossless compressed model. Its accuracy converged at 0.747 which is 0.02 larger than the lossless accuracy. Furthermore its size converged at a total size of 5519 contexts which is significantly less than the median size of the lossless predictors (figure 8.15). The reason for that is that the lossy algorithm does not add states whose predictions differ by less than $\varepsilon = 0.01$ from the existing prediction to avoid overfitting the model. Apparently, this turned out to be beneficial in the traces.

These results confirm that our compression methods can successfully be applied to the compression of a set of predictors as well. It even performed slightly better than the lossless compressed model with less than 75% of its size.

8.5.2 Continuous Variable Order Predictor

For the continuous variable order predictor, we introduced two compression variants: The size based compression which allowed larger predictors to expand only if other predictors inherently used less than average space, and the time based compression that compressed the model first storing the least recently updated prediction. The continuous variable order model requires much more space than the trained prediction models because it stores the transition probability function for each state. We compared the accuracy while using size limitations between 1500 and 10000 to the uncompressed predictor for both variants (figures 8.16 and 8.17). Overall, the size based approach performed better except for $L = 10000$ where it lost by a small margin. However, in the traces evaluated it had difficulties with the trace with the lowest accuracy which deviated more than 20 percentage points from the accuracy obversed in the uncompressed model. The size based approach only has the ability

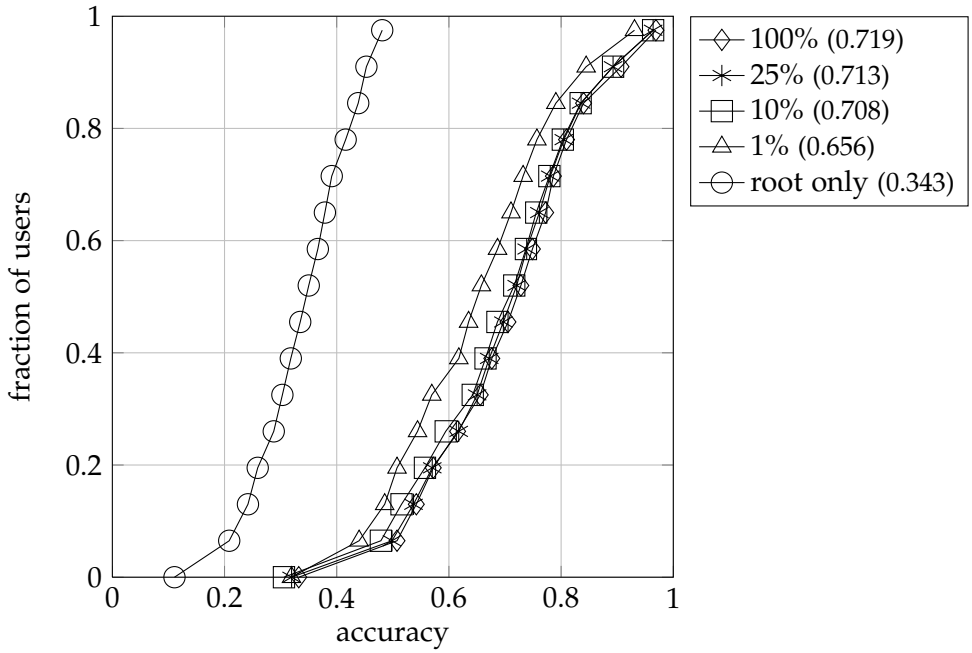

Figure 8.12: Accuracy for order 4 of the compressed continuous variable order predictor

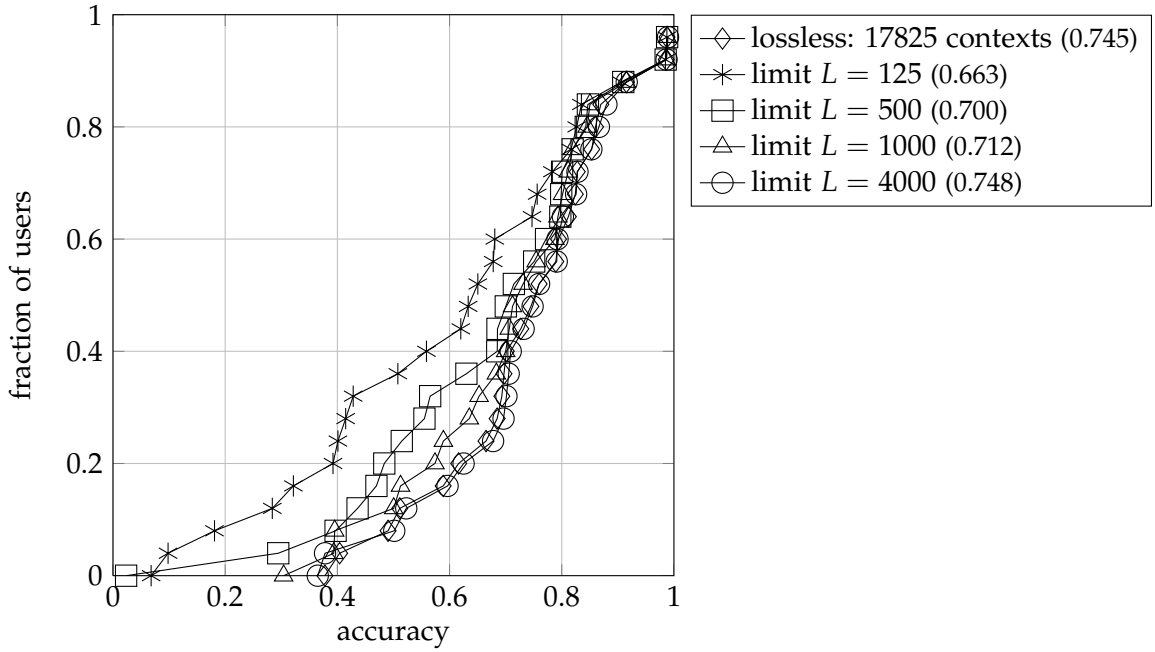

Figure 8.13: Accuracy for a set of 25 traces ($g_2(s)$)

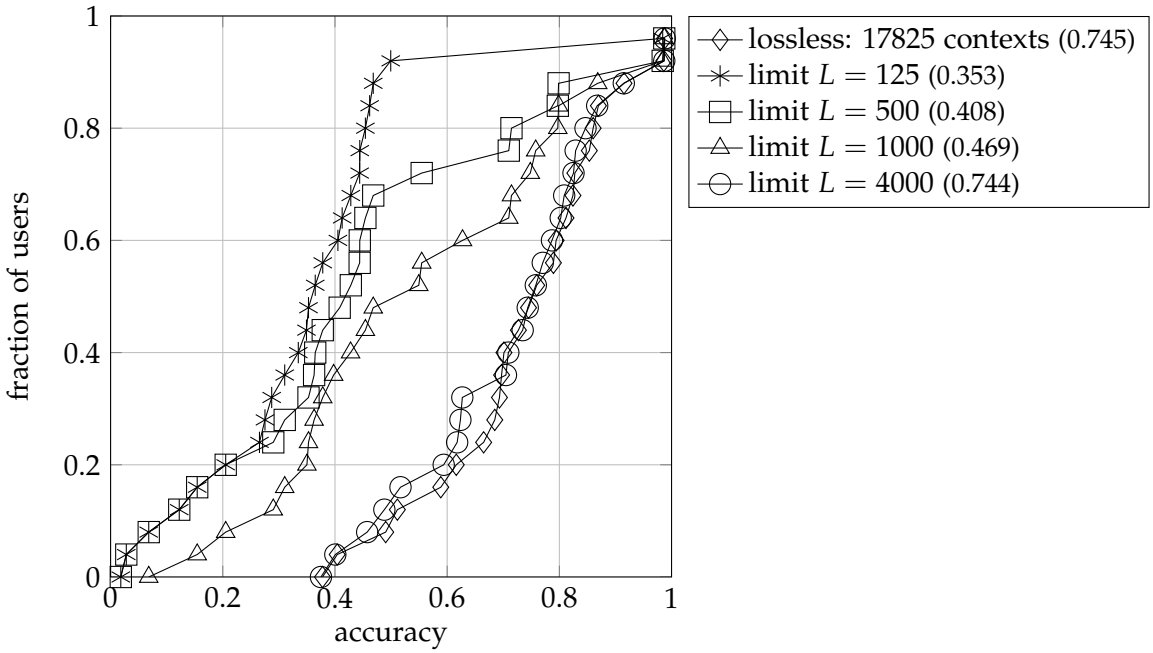

Figure 8.14: Accuracy for a set of 25 traces ($g_3(s)$)

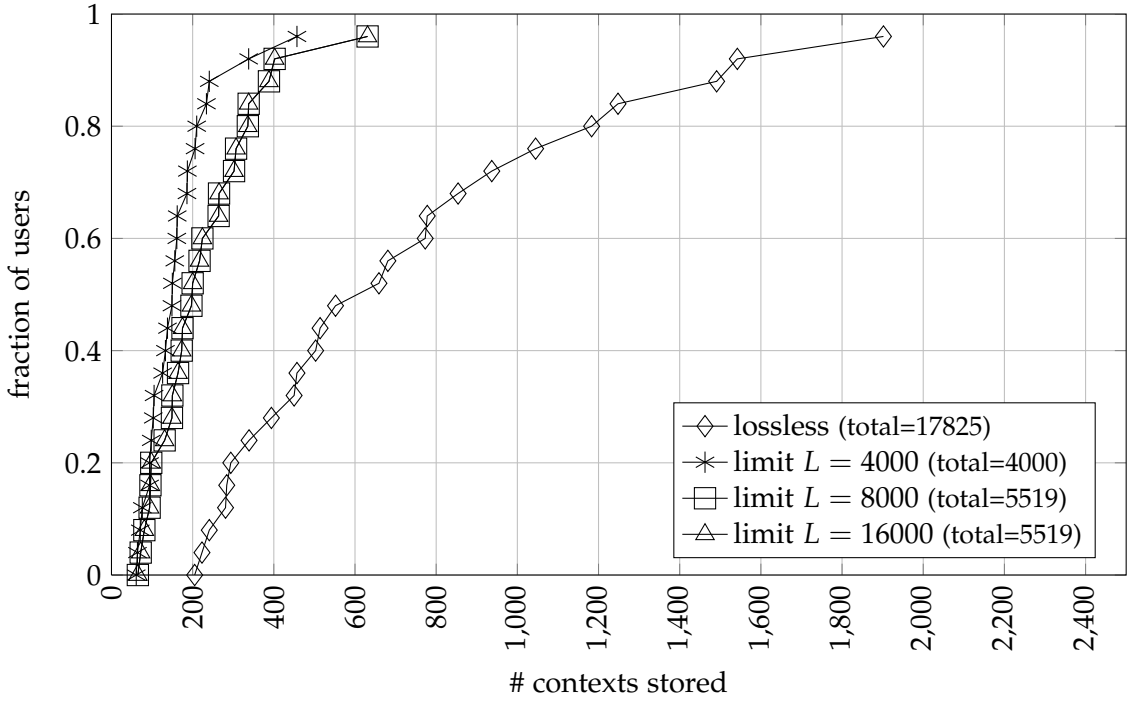

Figure 8.15: Total stored contexts for set of 25 traces ($g_2(s)$)

to adapt the sizes assigned to each predictor in case at least one predictor uses less size than average. A limit of 1500 means an average value of 60 contexts per predictor, which is easily reached when storing the whole transition probability function for each state. Therefore, in these cases the time based approach is preferable. When sufficient space is available, the size based adaption generally seems to be the better choice though.

8.6 Results from Generated Data

To see if the results are valid for other data too, we repeated some of the evaluations on a set of generated variable order traces with a maximum order of 4 and 7 different contexts.

8.6.1 Lossy Compression of Individual Models

First we evaluated how the ratings perform on generated data using a set of 200 generated traces with a training and evaluation sequence of length 5000 each (figure 8.18). The lossless variable order predictor had a median accuracy of 0.442 at a median size of 269. We compressed this model using ratings g_1, g_2 and g_3 to 10%. Using the real word traces we observed that the rating $g_1(s) = acc(s) * fre(s) * diff(s)$ was superior to the rating $g_2(s) = acc(s) * fre(s)$, in particular at small sizes. The order-based rating g_3 performed considerably worse. This still holds for the generated traces. The difference between g_1 and g_2 rated predictors is significant (0.01), but smaller than the difference to the predictor using rating g_3 (0.0299). Furthermore, in these data the g_1-rated predictor has the same median accuracy as the lossless predictor. They are not exactly equal since the average accuracy differs by around 0.005. The result is remarkable nevertheless because the size difference is significant as the compressed model only stores a median of 25 contexts while still achieving an accuracy equivalent to the lossless compressed predictor which requires 269 contexts.

8.6.2 Lossy Compression of Multiple Prediction Models

The compression of multiple models was evaluated by compressing a set of 25 prediction models using a trained predictor with rating g_2 and untrained predictors . The result for the trained variable order predictor (figure 8.19) resembles what we observed in the traces as well: At some point, the accuracy of the compressed model actually exceeded (0.451) the accuracy obtained using the lossless compressed model (0.448). We assume this is caused by the lossless variable order model not correctly modeling the variable order of the model underlying the data generation. The removal of some states by compressing the model actually made it better represent the generating process.

Because the training set and the evaluation set were generated in the same way, the untrained predictor perform quite similiar than the trained predictors. The difference between the time based and the size based approach is marginal (figures 8.20 and 8.21). However unlike in the real world traces, here the continuous predictor performs worse than the trained predictors

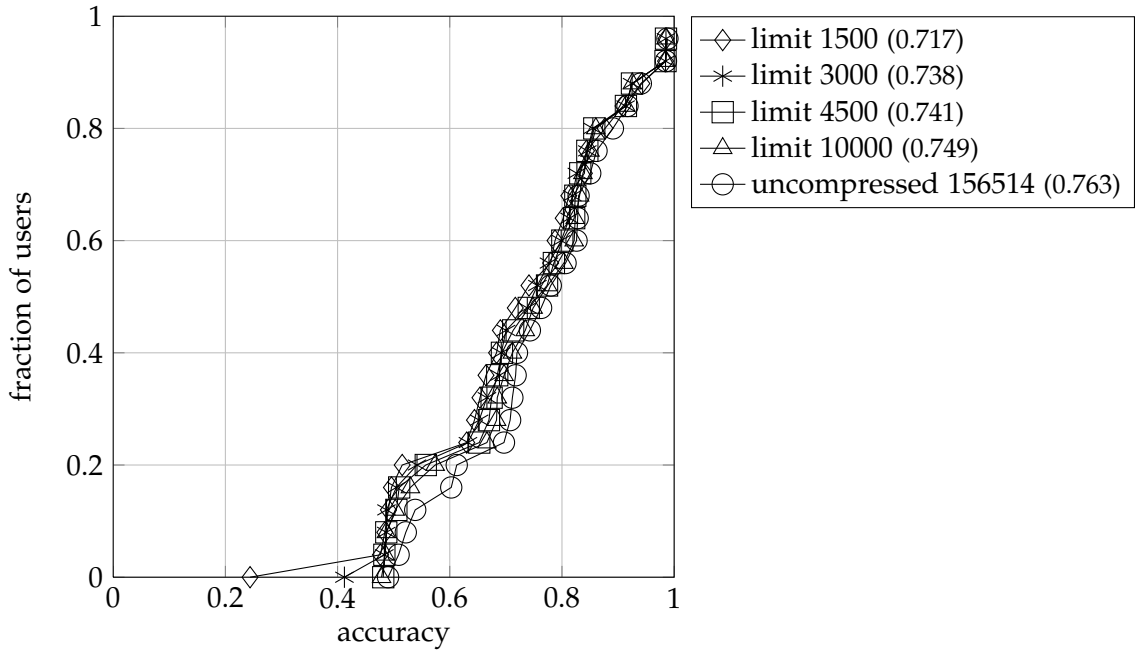

Figure 8.16: Accuracy for a set of 25 models with size based continuous compression

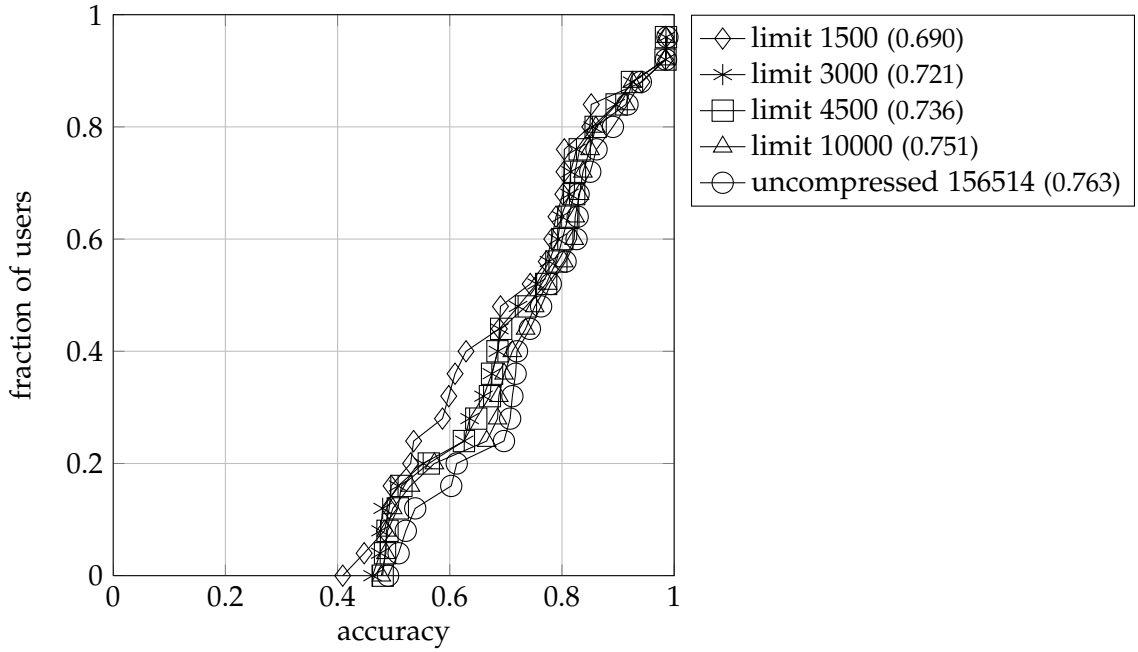

Figure 8.17: Accuracy for set of 25 models with time based continuous compression

because it first had to learn the transition probabilities and could not predict states not seen before. Conversely, this means that in real world data it is important to update the transition probabilities in regular intervals because the assumed underlying Markov model is changing constantly.

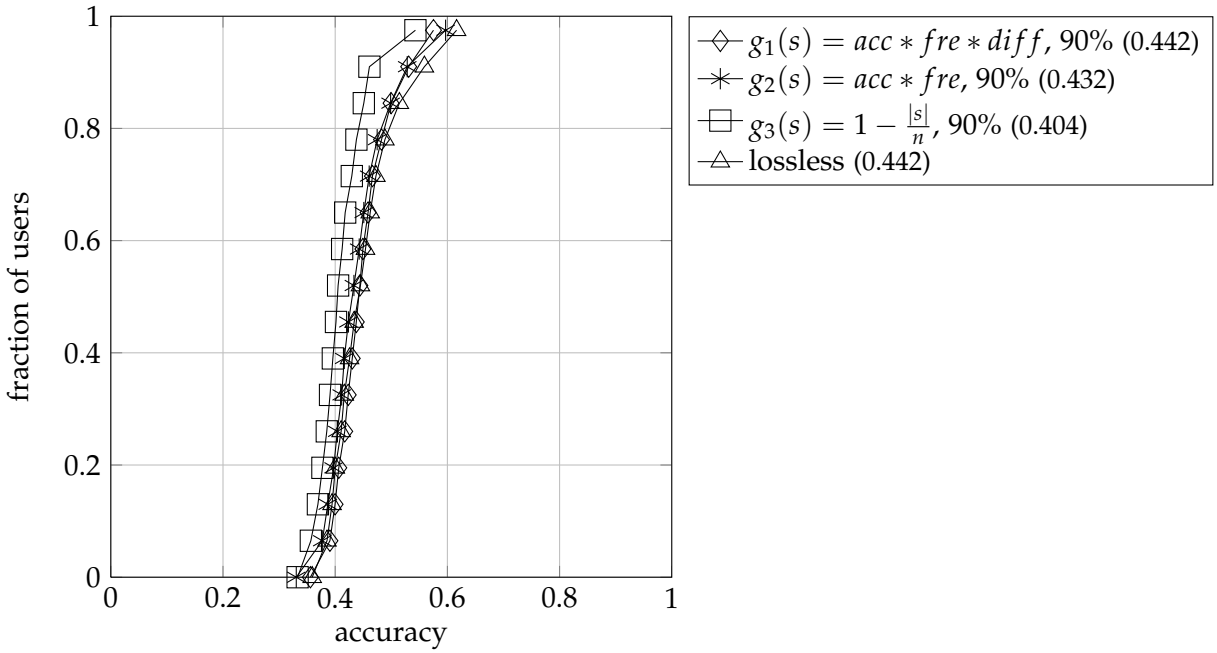

Figure 8.18: Accuracy for various ratings ($n = 4$ and 90% compression, generated data)

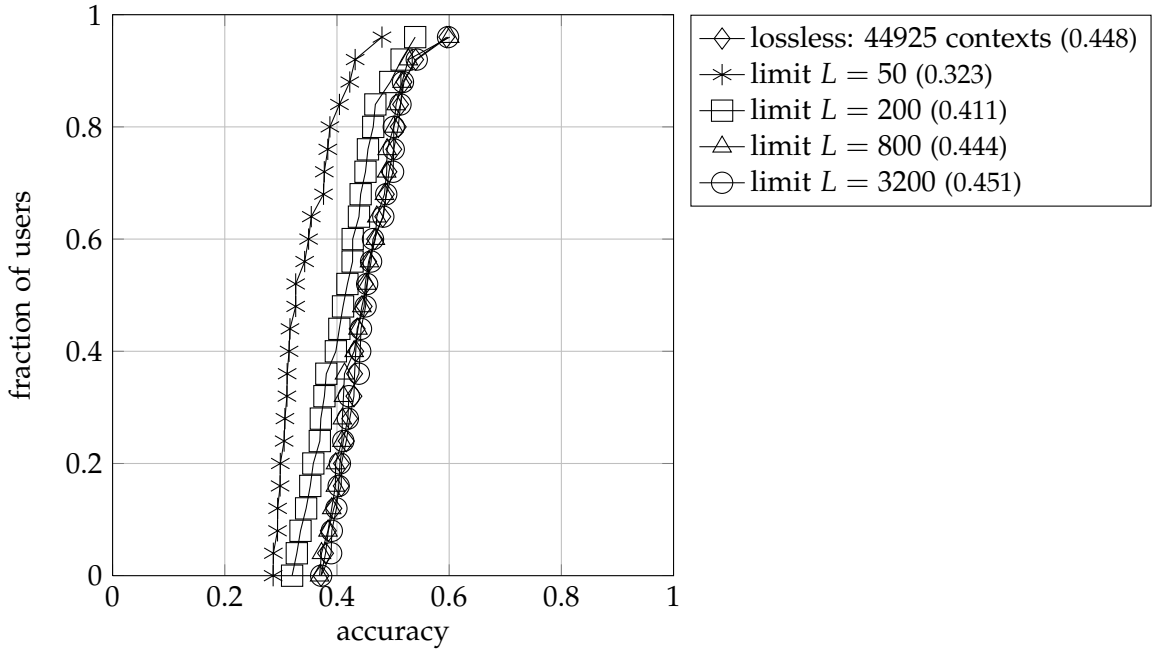

Figure 8.19: Accuracy for set of 25 traces ($g_2(s)$, generated data)

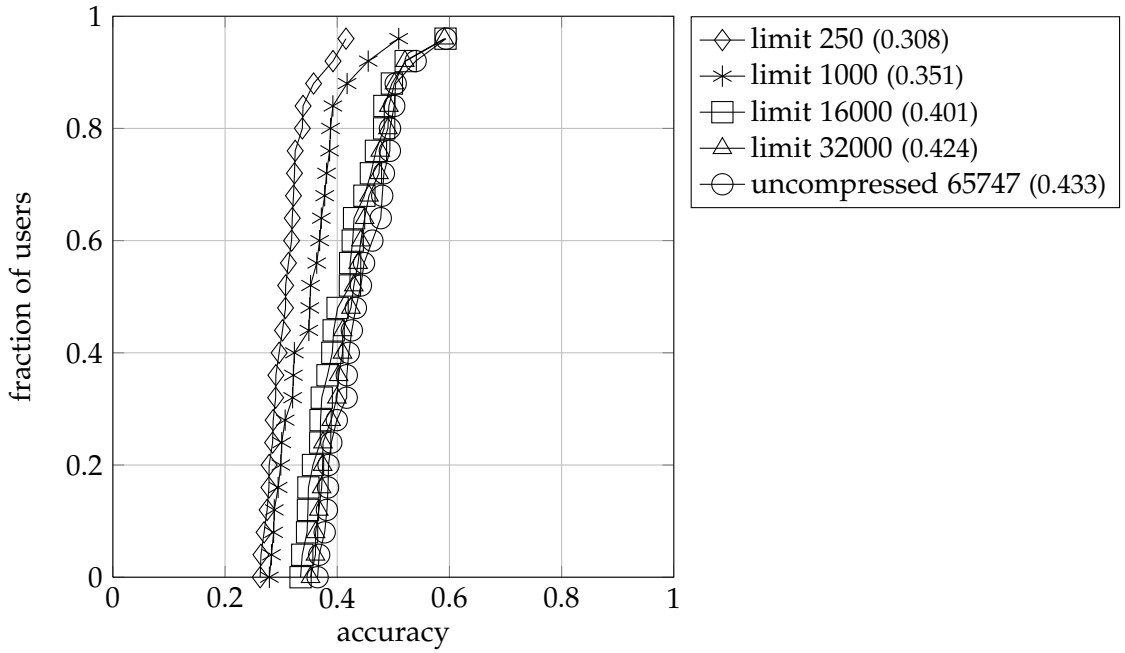

Figure 8.20: Accuracy for set of 25 models with size based compression, generated data

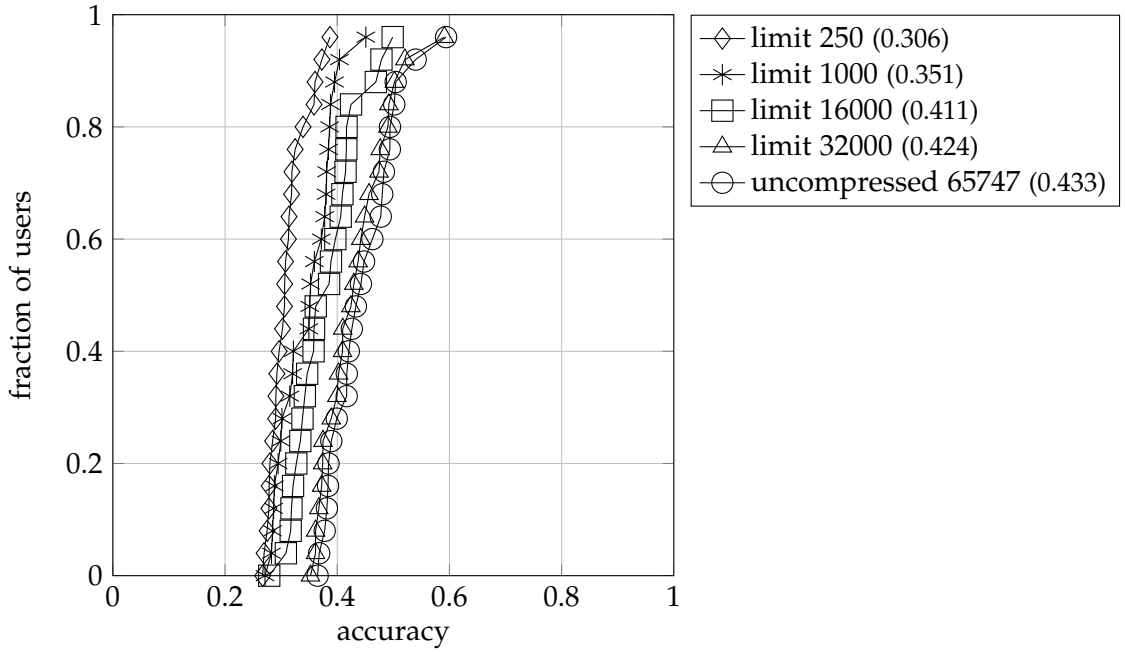

Figure 8.21: Accuracy for set of 25 models with time based compression, generated data

9 Conclusion and Outlook

The goal of this thesis was to develop methods in order to efficiently store prediction models on mobile devices. To this end, we presented approaches to compress prediction models when the allocated size is constrained. The evaluation showed that all compression mechanisms performed well both when compressing prediction models individually and as a set. In particular, they provided a better accuracy than achievable by compressing through decreasing the order of a prediction model. The lossless compression mechanism was able to reduce the size of Markov prediction models significantly by using a variable order model to store the prediction function. Moreover, by employing a lossless compression mechanism using ratings g_1 and g_2, the size of the lossless compressed model could be further reduced by more than 90% without a major impact on the accuracy. The difference between these ratings was marginal except for very high compression ratios when the $diff$ measure tipped the balance in favour of g_1. Due to the computational effort to update the $diff$ rating repeatedly, we could therefore conclude that it is only advisable to include it when a very strong compression is to be achieved. Our approach to limit continuous Markov models in size without losing the benefit of constant learning also was able to reduce the size with only a minor impact on the prediction accuracy. Our approaches for single model compression could be successfully transferred to the compression of multiple models, where the results were equally remarkable at high compression ratios and in some cases even outperformed the lossless compressed predictor. We can therefore summarize that our algorithms provide an efficient way to build and store prediction models under a size constraint.

Nevertheless, we see further possibilities to effectively reduce the size of the model that needs to be stored on a mobile system. The approach we suggested perceives a context as a whole. However, in real world applications, contexts usually consist of many different attributes such as the time of day, the location, the environment and the activity. It is possible that a subset of these attributes is sufficient to make a prediction, thus the prediction is independent of some attributes. In our approach these attributes are still accounted for individually in the prediction suffix tree. If we could identify those attributes, this would enable us to compress a prediction suffix tree by merging nodes that are equal short of these attributes. Another way to save space by exploiting the context itself is to build separate prediction models for different attributes which rarely change. For example, people usually have a different daily routine on working days and on weekends. Splitting the publisher's prediction model into a 'workday-model' and a 'weekend-model', it would suffice to store only one of them at a time on the consumer's device. Of course the publisher or the server would have to transmit a new model to the consumer every few days, but this could be done while using a wired connection or a wireless connection with a low energy profile.

In this thesis, we covered Markov models without a notion of time. Hence, the predictions were only concerned with what will happen next and not with when it will happen. But information about time is often equally important in real world applications. In order to make predictions such as how long the current context will be held and when a specific context is about to be seen next, another prediction model such as the semi Markov model has to be used. With some modifications such as an adjusted rating function, we think concepts can also be applied to these predictors.

Bibliography

[Bar04] J. E. Bardram. Applications of context-aware computing in hospital work: examples and design principles. In *Proceedings of the 2004 ACM symposium on Applied computing*, SAC '04, pp. 1574–1579. ACM, New York, NY, USA, 2004. (Cited on page 9)

[BDR07] M. Baldauf, S. Dustdar, F. Rosenberg. A survey on context-aware systems. *Int. J. Ad Hoc Ubiquitous Comput.*, 2:263–277, 2007. (Cited on page 11)

[BEYY04] R. Begleiter, R. El-Yaniv, G. Yona. On prediction using variable order Markov models. *JOURNAL OF ARTIFICIAL INTELLIGENCE RESEARCH*, 22:385–421, 2004. (Cited on page 12)

[BI04] L. Bao, S. S. Intille. Activity recognition from user-annotated acceleration data. pp. 1–17. Springer, 2004. (Cited on page 11)

[CIW84] J. G. Cleary, Ian, I. H. Witten. Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Transactions on Communications*, 32:396–402, 1984. (Cited on page 12)

[CRA] CRAWDAD. Darthmouth College Movement Traces 2001-2004. http://crawdad.cs.dartmouth.edu/meta.php?name=dartmouth/campus/movement. (Cited on page 41)

[DA00] A. K. Dey, G. D. Abowd. Towards a Better Understanding of Context and Context-Awareness. In *Workshop on The What, Who, Where, When, and How of Context-Awareness, as part of the 2000 Conference on Human Factors in Computing Systems (CHI 2000)*. 2000. (Cited on pages 11 and 13)

[Fin07] G. A. Fink. *Markov Models for Pattern Recognition: From Theory to Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. (Cited on page 17)

[Fin11] T. Fink. *Effiziente vorhersage-basierte Verteilung von Kontext-Informationen in mobilen Systemen*. Master's thesis, Universität Stuttgart, 2011. (Cited on page 9)

[How71] R. A. Howard. *Dynamic Probabilistic Systems*. John Wiley, New York, 1971. (Cited on page 17)

[IBB03] S. S. Intille, L. Bao, L. Bao. Physical Activity Recognition from Acceleration Data under SemiNaturalistic Conditions. Technical report, 2003. (Cited on page 11)

[ISo03] J. Indulska, P. Sutton. Location management in pervasive systems. In *Proceedings of the Australasian information security workshop conference on ACSW frontiers 2003 - Volume 21*, ACSW Frontiers '03, pp. 143–151. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2003. (Cited on page 11)

[KKP+03] P. Korpipää, M. Koskinen, J. Peltola, S. M. Mäkelä, T. Seppänen. Bayesian approach to sensor-based context awareness. *Personal Ubiquitous Comput.*, 7(2):113–124, 2003. (Cited on page 11)

[KL51] S. Kullback, R. A. Leibler. On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951. (Cited on page 29)

[KMo5] D. Katsaros, Y. Manolopoulos. A suffix tree based prediction scheme for pervasive computing environments. In *In Lecture notes in computer science (LNCS*, pp. 267–277. Springer-Verlag, 2005. (Cited on page 11)

[Mö3] J. Mäntyjärvi. *Sensor-based context recognition for mobile applications*. Ph.D. thesis, University of Oulu, Finland, 2003. (Cited on page 11)

[MPF+] M. Musolesi, M. Piraccini, K. Fodor, A. Corradi, A. T. Campbell. Supporting Energy-Efficient Uploading Strategies for Continuous Sensing Applications on Mobile Phones. (Cited on page 12)

[MW] Merriam Webster Online Dictionary. http://www.merriam-webster.com/. (Cited on page 9)

[Pas98] M. J. Pascoe. Adding Generic Contextual Capabilities to Wearable Computers. In *Proceedings of the 2nd IEEE International Symposium on Wearable Computers*, ISWC '98, pp. 92–. IEEE Computer Society, Washington, DC, USA, 1998. (Cited on page 9)

[PBTU] J. Petzold, F. Bagci, W. Trumler, T. Ungerer. Comparison of Different Methods for Next Location Prediction. (Cited on page 9)

[RN10] S. J. Russell, P. Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010. (Cited on page 17)

[ROo8] U. Rathnayake, M. Ott. Predicting network availability using user context. In *Proceedings of the 5th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*, Mobiquitous '08, pp. 49:1–49:8. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, 2008. (Cited on pages 9 and 12)

[RST96] Ron, Singer, Tishby. The Power of Amnesia: Learning Probabilistic Automata with Variable Memory Length. *MACHLEARN: Machine Learning*, 25, 1996. (Cited on pages 12, 23, 29, 30 and 31)

[SAW94] B. Schilit, N. Adams, R. Want. Context-Aware Computing Applications. In *Workshop on Mobile Computing Systems and Applications*. Santa Cruz, CA, U.S., 1994. (Cited on pages 9 and 11)

[SBG98] A. Schmidt, M. Beigl, H. w. Gellersen. There is more to Context than Location. *Computers and Graphics*, 23:893–901, 1998. (Cited on page 11)

[SKJH04] L. Song, D. Kotz, R. Jain, X. He. Evaluating Location Predictors with Extensive Wi-Fi Mobility Data. In *In Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM*, pp. 1414–1424. 2004. (Cited on page 11)

[SMR+97] T. Starner, S. Mann, B. Rhodes, J. Levine, J. Healey, D. Kirsch, R. Picard, A. Pentland. Augmented reality through wearable computing. *PRESENCE*, 6:386–398, 1997. (Cited on page 9)

[ST94] B. Schilit, M. Theimer. Disseminating Active Map Information to Mobile Hosts. *IEEE Network*, 8:22–32, 1994. (Cited on pages 9 and 11)

[Wei91] M. Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):66–75, 1991. (Cited on page 11)

[WST95] F. M. J. Willems, Y. M. Shtarkov, T. J. Tjalkens. The Context-Tree Weighting Method: Basic Properties. *IEEE Trans. Inform. Theory*, 41:653–664, 1995. (Cited on page 12)

[Zab89] S. L. Zabell. The Rule of Succession. *Erkenntnis (1975-)*, 31(2/3):pp. 283–321, 1989. (Cited on page 18)

[ZL78] J. Ziv, A. Lempel. Compression of Individual Sequences via Variable-Rate Coding, 1978. (Cited on pages 11 and 12)

[ZL07] A. Zimmermann, A. Lorenz. A.: An operational definition of context. *In: CONTEXT*, 2007. (Cited on page 11)

All URLs were last accessed on January 4, 2012.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Jörg Belz)