ASSIGNMENT OF BACHELOR’S THESIS

Title: Parallel implementation of dynamic naive Bayesian classifier
Student: Pavel Lučivňák
Supervisor: Ing. Tomáš Šabata
Study Programme: Informatics
Study Branch: Computer Science
Department: Department of Theoretical Computer Science
Validity: Until the end of summer semester 2018/19

Instructions

A dynamic naive Bayes classifier (DNBC) is an extension of the popular probabilistic graphical model called hidden Markov model. Output variables are assumed to be statistically independent, which helps us in case of the curse of dimensionality occurring in high-dimensional space. Moreover, output variables that come from different probability distributions can be learned easier.

This thesis aims to create a parallel implementation of DNBC that can be easily executed on a computational cluster. For that reason, the algorithm will be implemented in the Scala language on top of Apache Spark.

1) Study DNBC and its possibilities of parallelism.
2) Implement DNBC that is applicable for both discrete and continuous output variables.
3) Parallelize the implementation on top of Apache Spark.
4) Evaluate the parallel implementation and compare it with the sequential implementation in terms of parallel scalability.

References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague January 15, 2018
Bachelor’s thesis

Parallel implementation of dynamic naive Bayesian classifier

Pavel Lučivňák

Department of Theoretical Computer Science
Supervisor: Ing. Tomáš Šabata

May 9, 2018
I would like to thank my family that has supported me during my work on this thesis. A special thanks to Ing. Tomáš Šabata for offering this interesting thesis topic and for leading me throughout the work.

Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum provided under the programme ”Projects of Large Research, Development, and Innovations Infrastructures” (CESNET LM2015042), is greatly appreciated.
Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the “Work”), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 9, 2018
Czech Technical University in Prague  
Faculty of Information Technology  
© 2018 Pavel Lučivňák. All rights reserved.  
*This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author’s permission is prohibited (with exceptions defined by the Copyright Act).*

**Citation of this thesis**

Abstrakt


Klíčová slova  dynamický naivní Bayesovský klasifikátor, skrytý Markovův model, metoda maximální věrohodnosti, Scala, Apache Spark

Abstract

Dynamic naive Bayesian classifier (DNBC) has many applications, such as in speech recognition, handwriting recognition or weather prediction. DNBC
extends a hidden Markov model by supporting multiple observed variables. It is assumed that these variables are mutually statistically independent. This assumption greatly simplifies computations and a phenomenon called *curse of dimensionality* does not occur. I have implemented the classifier in Scala language on top of Apache Spark. The implementation can be parallelized by using Map Reduce paradigm. I have managed to double the speed up when using 15 processor cores. I have further demonstrated, that the speed up can be achieved not only by increasing the number of cores, but also by increasing the number of machines in a cluster.

**Keywords** dynamic naive Bayesian classifier, hidden Markov model, maximum likelihood estimation, Scala, Apache Spark
Contents

Introduction 1

1 Mathematical background 3
   1.1 Probability density function 3
   1.2 Gaussian distribution 3
   1.3 Gaussian mixture distribution 4
   1.4 Summary 5

2 Parameter estimation 7
   2.1 Maximum likelihood estimation 7
   2.2 Discrete distribution 7
   2.3 Gaussian distribution 8
   2.4 Gaussian mixture distribution 10

3 Analysis 13
   3.1 Markov chain 13
   3.2 Hidden Markov model 15
   3.3 Bayesian network 18
   3.4 Naïve Bayesian classifier 19
   3.5 Dynamic Bayesian network 21
   3.6 Dynamic naïve Bayesian classifier 21

4 Implementation 27
   4.1 Apache Spark 27
   4.2 Project structure 28
   4.3 Usage 28
   4.4 Underflow 31
   4.5 Data set generation 31
   4.6 Source code 32
List of Figures

1.1 Visualization of Gaussian PDF with parameters $\mu = 10$ and $\sigma^2 = 3^2$. 4
1.2 Visualization of a mixture with three components with parameters
$\mu_1 = 5, \mu_2 = 10, \mu_3 = 15$ and $\sigma^2_{1,2,3} = 2^2$. Weights are $w_{1,2,3} = \frac{1}{3}$. 5

2.1 Frequency of data in a discrete data set. 8
2.2 Likelihood function $L$ of data in the discrete data set. Likelihood is zero at undefined states. 8
2.3 Histogram of randomly generated data from normal distribution $\mathcal{N}(40,32^2)$. The green curve is a plot of normal distribution with the maximum likelihood estimate of $\theta$ parameters. 9
2.4 Histogram of randomly generated data from two normal distributions $\mathcal{N}(10,5^2)$ and $\mathcal{N}(30,4^2)$. The green curve is the best estimate, found using EM algorithm, of underlying normal mixture distribution. Dotted curves are estimates of individual Gaussian mixture components. 10

3.1 Visualization of a Markov chain with three states. 14
3.2 Visualization of a walk through a hidden Markov model. At each time point, there is a hidden state $y_t$ and an observed symbol $x_t$. 16
3.3 Visualization of a hidden Markov model with three states and a continuous observed variable. The gray arrows represent the initial probability vector. 19
3.4 Visualization of a Bayesian network with three states. 20
3.5 Visualization of a dynamic Bayesian network with four variables at time $t$. 21
3.6 Visualization of a dynamic naive Bayesian classifier with two observed variables. At each time point, there is a hidden state $y_t$ and two observed states: $x^1_t$ and $x^2_t$. 22

4.1 Dependencies among modules in $dnbc$-scala project. 28
5.1 Visualization of a Toy Robot data set [2].

5.2 Visualization of the Gaussian mixture data set. The solid curves represent a distribution of the observed continuous variable given hidden state is false. If the hidden state is true, the variable follows a Gaussian distribution as visualized by the dotted curve.

5.3 Speed up ratio per number of workers.

5.4 Continuous emissions speed up ratio per number of workers.

5.5 Continuous emissions speed up ratio per number of processors per worker.

5.6 Learning time per number of training sequences.

5.7 Learning time per number of hidden states.

5.8 Learning time per number of observed discrete variables.
List of Tables

3.1 CPT for dehydration .................. 20
3.2 CPT for cancer .......................... 20
3.3 CPT for red skin ....................... 20

5.1 Results on realistic data sets. There are two main reasons for relatively low success rates on the first two data sets: only 90% of observed symbols are accurate and there are often multiple transitions to hidden states with the same observed symbol. ............ 35
5.3 Base data set generation and evaluation parameters. ............ 36
Hidden Markov model (HMM) is a probabilistic model that is widely used in many fields, e.g. speech recognition [4], handwriting recognition [5] or weather prediction [6]. The model considers a hidden random variable and an observed random variable. For example, in the case of speech recognition, the hidden variable would be an actual letter from the alphabet a person said. The observed variable could then be a tone pitch. To estimate the current letter of a word that the person said, the model takes into account the previously said letter and the current pitch level.

The model is first trained using labeled data—containing both hidden states and observed states [1]. Then an inference can be used to predict hidden states, knowing only observed states.

A limitation of HMM is that it only supports one observed random variable. In case of the speech recognition, there are more variables than just the tone pitch that are relevant to letter estimation. One could increase the dimensionality of the observed variable. However, this would lead to a phenomenon called curse of dimensionality that would result in model training to be intractable.

Dynamic naive Bayesian classifier (DNBC) addresses this issue by supporting multiple observed variables. To avoid the curse of dimensionality, the observed variables are assumed to be statistically independent with each other (given the hidden variable). Due to the fact that the observed variables are statistically independent, the model can be easily trained in the parallel fashion.

However, there are no existing DNBC libraries written in Scala language. Although there exists a library for Bayesian networks [7], which is a superset of DNBC, it does not support parallelization on top of Apache Spark. Since Bayesian networks are above the scope of this thesis, I have decided to implement DNBC by myself.

---

1This is true for maximum likelihood estimation learning method. In the case of Baum-Welch algorithm, only observed states and list of possible hidden states are necessary.
Introduction

In the first chapter, I briefly cover preliminary mathematical background used in this thesis—namely Gaussian distribution and Gaussian mixture distribution. Then, in the second chapter, I discuss parameter estimation of probabilistic models. This will be useful for training of HMM and DNBC later on.

In the third chapter, Analysis, I introduce foundations of DNBC. First, a Markov chain is introduced together with hidden Markov model. Afterwards, Bayesian networks and its derivatives, naive Bayesian classifier and dynamic Bayesian networks, are discussed. This gives us enough foundation for defining dynamic naive Bayesian classifier.

DNBC is analyzed in terms of three basic operations that can be performed on it—learning, inference and scoring. I also provide an application example and discuss limitations of DNBC.

In the fourth chapter, implementation details are described. Design decisions I made when creating a DNBC library are covered as well.

Finally, in the fifth chapter, I show how well DNBC performs in terms of the inference accuracy on realistic data sets. In the end, I show experimental results on randomly generated data sets. The experiments are focused on learning time and scalability.
Mathematical background

In this chapter I cover mathematical topics that are necessary for proper understanding of the following chapters. These topics are part of probability theory. I suggest taking a look at Introduction to Probability [8] for details.

1.1 Probability density function

Probability density function (PDF) of a continuous random variable determines a likelihood that a given value occurs. Furthermore, \( P(a \leq x \leq b) = \int_{a}^{b} \text{PDF}(x)dx \) (1.1)

To satisfy the property that \( P(-\infty < x < \infty) = 1 \), the following has to hold:

\[
\int_{-\infty}^{\infty} \text{PDF}(x)dx = 1.
\]

1.2 Gaussian distribution

Gaussian (normal) distribution is defined by a probability density function

\[
\text{PDF}_{\mathcal{N}(\mu,\sigma^2)}(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.
\]

The distribution is defined by two parameters—mean \( \mu \) and variance \( \sigma^2 \). Figure 1.1 provides a visualization of the probability density function.

Normal distribution is one of the most widespread distribution found in nature [9]. Imagine \( n \) independent, identically distributed (i.i.d.) random variables. For large \( n > 30 \), both sum and average of the variables converge
1. Mathematical background

Figure 1.1: Visualization of Gaussian PDF with parameters $\mu = 10$ and $\sigma^2 = 3^2$.

to normal distribution. This phenomenon occurs because of the Central Limit Theorem. See book *Introduction to Probability* \[8\] for a proof.

1.3 Gaussian mixture distribution

Gaussian mixture distribution is a weighted sum of multiple components that follow normal distribution. The PDF is given by \[1.2\] where $n$ is number of components and $w_i$ is a weight of particular component.

$$\text{PDF}_{\mathcal{N}(\theta)}(x) = \sum_{i=1}^{n} w_i \text{PDF}_{\mathcal{N}(\mu_i,\sigma_i^2)}(x)$$ \hspace{1cm} \text{(1.2)}$$

The following conditions have to hold:

$$w_i \geq 0, 1 \leq i \leq n,$$

$$\sum_{i=1}^{n} w_i = 1.$$
1.4 Summary

Both Gaussian distribution and Gaussian mixture distribution are widely used in HMM and DNBC to represent continuous variables.

Figure 1.2: Visualization of a mixture with three components with parameters $\mu_1 = 5, \mu_2 = 10, \mu_3 = 15$ and $\sigma^2_{1,2,3} = 2^2$. Weights are $w_{1,2,3} = \frac{1}{3}$. Author: Smason79 [I].
Parameter estimation

In this chapter, I briefly introduce parameter estimation that is commonly used in learning of DNBC. The goal of parameter estimation is to estimate parameters of a probability distribution in a way that describes given data the best.

2.1 Maximum likelihood estimation

Maximum likelihood estimation (MLE) is a technique for finding parameters of a probabilistic distribution that best describe a random variable. The method aims to maximize the likelihood of all the learning data.

Formally arg\(\max_{\theta \in \Theta} \prod_{j=1}^{M} L(x_j, \theta)\), where:

- \(M\) is number of data points,
- \(X = \{x_1, x_2, \ldots, x_M\}\) is a learning set of data points,
- \(L\) is a likelihood function (defined further),
- \(\theta\) describes model parameters,
- \(\Theta\) is a model parameters space.

2.2 Discrete distribution

Here I present a straightforward way of defining the likelihood function in case of discrete random variables. \(L(x, \theta) = x_{cnt} / M\), where \(x_{cnt}\) is the number of times \(x \in X\) occurs in learning data. Since the likelihood function does not depend on parameter \(\theta\), there is no expression to optimize.

There is a reason I did not choose any standard discrete probability distribution to define \(L\). The random variable does not have to be \(\mathbb{Z}\), nor \(\mathbb{N}\). In fact it can be any abstract object, where comparison between two objects doesn’t
2. Parameter estimation

Figure 2.1: Frequency of data in a discrete data set.

Figure 2.2: Likelihood function $L$ of data in the discrete data set. Likelihood is zero at undefined states.

make sense. Therefore, it wouldn’t make sense to assign non-zero probability to values that are not specified in learning phase.

As an example, consider the following data: \{dog, bird, dog, cat, bird, dog\}. Figure 2.2 shows a likelihood function $L$ associated with the data.

2.3 Gaussian distribution

The normal distribution is defined by $\theta = \{\mu, \sigma^2\}$. In MLE, the goal is to find parameter $\theta \in \Theta$, such that the product in (2.1) is maximized.

$$M \prod_{j=1}^{M} L(x_j, \theta)$$  (2.1)
2.3. Gaussian distribution

In the case of Gaussian distribution, the likelihood function $L$ is defined by

$$L(x, \theta) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where $x \in X$. Taking derivative of 2.1 with respect to $\mu$ equal to 0 yields

$$\sum_{j=1}^{M} x_j - M\mu = 0.$$

Maximum likelihood estimate of $\mu$ is therefore

$$\mu_{\text{est}} = \overline{X}_M.$$

Derivative of 2.1 with respect to $\sigma^2$ equal to zero results in MLE of $\sigma^2$ to be

$$\sigma^2_{\text{est}} = \frac{1}{M} \sum_{j=1}^{M} (x_j - \overline{X}_M)^2.$$

Figure 2.3 shows an example of MLE on a normally distributed random variable.


2. Parameter estimation

Figure 2.4: Histogram of randomly generated data from two normal distributions $\mathcal{N}(10, 5^2)$ and $\mathcal{N}(30, 4^2)$. The green curve is the best estimate, found using EM algorithm, of underlying normal mixture distribution. Dotted curves are estimates of individual Gaussian mixture components.

2.4 Gaussian mixture distribution

Given the number of components $K$ components, there are component parameters $\theta_1, \theta_2, \ldots, \theta_K$ and weights $w_1, w_2, \ldots, w_K$ to estimate. In this section, I provide a description of expectation-maximization algorithm based on the lecture notes by Padhraic Smyth [10]. This algorithm provides an estimate of the aforementioned parameters.

Let’s define a membership weight $w_{jk}$, of data point $x_j$ in component $k$ as

$$w_{jk} = \frac{\alpha_k \text{PDF}_N(\theta_k)(x_j)}{\sum_{i=1}^K \alpha_i \text{PDF}_N(\theta_i)(x_j)},$$

where $\alpha_k$ is a probability that a randomly selected $x_j$ was generated by the component $k$. $w_{jk}$ can be thought of as a level of certainty that $x_j$ was generated by the component $k$. $\sum_{k=1}^K w_{jk} = 1$ holds.

The likelihood of all data points given Gaussian mixture parameters is defined as

$$l = \sum_{j=1}^M \sum_{k=1}^K \alpha_k \text{PDF}_N(\theta_k)(x_j).$$

(2.2)
2.4. Gaussian mixture distribution

EM is an iterative algorithm, typically iterating until convergence is detected. Each iteration consists of two parts, an E-step and an M-step. In the E-step, weight $w_{jk}$ is computed for each data point $x_j$ and component $k$. Define $W_k = \sum_{j=1}^{M} w_{jk}$, the weight of all data points in component $k$. In the M-step, new model parameters are calculated as follows:

$$\alpha_{k}^{\text{new}} = \frac{W_k}{M},$$

$$\mu_{k}^{\text{new}} = \frac{\sum_{j=1}^{M} w_{jk} x_j}{W_k},$$

$$\sigma_{k}^{2\text{new}} = \frac{\sum_{j=1}^{M} w_{jk} (x_j - \mu_{k}^{\text{new}})^2}{W_k},$$

where $\mu_{k}^{\text{new}}$ resembles standard equation for estimating mean, and $\sigma_{k}^{2\text{new}}$ is similar to an equation for estimating variance. The differences are in weighting.

In the beginning of the algorithm, an initial guess of model parameters is necessary. This can be chosen randomly or through a heuristic. Termination of the algorithm is determined by checking that the likelihood 2.2 of all data points hasn’t improved enough in between iterations. In other words

$$l_{\text{cur}} - l_{\text{prev}} < \text{tol}.$$

One issue with the EM algorithm is that it does not guarantee to find a global optimum [10].
In this chapter I introduce models that are fundamental for understanding DNBC. I begin with a description of a basic probabilistic model—Markov chain. Then I focus on more advanced models such as HMM and Bayesian networks.

3.1 Markov chain

Markov chain describes possible sequences of events in which a probability of being at a state at given time depends only on value of previous state.

3.1.1 Description

Markov chain is defined by

- a set of states $S = \{S_1, S_2, \ldots, S_N\}$,
- a transition matrix $A \in \mathbb{R}^{N \times N}$,
- an initial probability vector $\pi \in \mathbb{R}^N$.

Where $a_{ij}$, $1 \leq i, j \leq N$ is a probability of transitioning from state $S_i$ into state $S_j$. $\pi_i$ is a probability that the initial state is $S_i$.

Suppose a sequence of states $y_1, y_2, \ldots, y_T$, where $T \geq 2$. At time $t$, where $1 \leq t \leq T - 1$, the transition probability $a_{ij}$ is defined as $P(y_{t+1} = S_j | y_t = S_i)$.

3.1.2 Markov property

Markov chain satisfies a Markov property. This property says that $P(y_{t+1} = S_i | y_1 = s_1, y_2 = s_2, \ldots, y_t = s_t) = P(y_{t+1} = S_i | y_t = s_t)$. In other words, being at state $S_i$ at time $t + 1$ only conditionally depends on being at state $s_t$ at time $t$. Further history doesn’t influence the probability.
3. Analysis

![Markov chain diagram]

Figure 3.1: Visualization of a Markov chain with three states.

3.1.3 Example

As an example, consider a Markov chain (MC) with three states: $S_1 = \text{sunny}$, $S_2 = \text{rainy}$ and $S_3 = \text{cloudy}$. This MC models how weather changes every day. Suppose the following transition matrix:

$$A = \begin{bmatrix} 0.6 & 0.1 & 0.3 \\ 0.1 & 0.7 & 0.2 \\ 0.3 & 0.3 & 0.4 \end{bmatrix}. $$

The transition matrix is readable for a human. For example, if the weather is sunny, tomorrow will be most likely sunny as well. With probability 0.3, it will be cloudy. Rainy weather is very unlikely tomorrow. Consider the following initial probability vector:

$$\pi = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.5 \end{bmatrix}. $$

As it can be seen, the most likely weather on first day of measurement is cloudy.
3.1.4 Limitations

The previous example only takes into account the weather at time $t$ to predict the weather at time $t+1$. In the reality, there are multiple factors that influence what the weather will be like. According to the example, if a weather is sunny now, it is relatively unlikely it will start raining. If we, however, take into account a humidity, we can improve the prediction. If we know that the humidity is high, even though it was sunny, we can predict it started raining. Hidden Markov model addresses this issue.

3.2 Hidden Markov model

Hidden Markov model (HMM) is an extension of Markov chain. The states are now called hidden states and every hidden state has an associated observed symbol.

3.2.1 Description

HMM is defined by (based on paper by L. Rabiner [4]):

- a set of hidden states $S = \{S_1, S_2, \ldots, S_N\}$,
- a set of observed symbols $X$,
- a transition matrix $A \in \mathbb{R}^{N \times N}$,
- an observation probability functions $B = \{b_1(o), b_2(o), \ldots, b_N(o)\}$,
- an initial probability vector $\pi \in \mathbb{R}^N$.

The values of the $X$ can be either discrete or continuous. The values of $S$ are assumed to be discrete. The observation probability function $b_i(o)$ describes a probability $P(x_t = o|y_t = S_i)$, where $o \in X$. This probability function is also called an emission function.

Imagine a walk through a HMM graph (see figure 3.3 for an example graph). At each time point $t$, there is a hidden state $y_t$ and an observed symbol $x_t$. A value of the observed symbol conditionally depends on a value of the hidden state. The dependency is described by emission function $b_i(o)$. If $t \neq 1$ and $T > 1$, there is a hidden state $y_t$ that conditionally depends on a value of previous hidden state $y_{t-1}$. The dependency is described by the transition $a_{ij}$. If $t = 1$, there is a hidden state $y_1$ that conditionally depends on the initial transition $\pi_i$.
3. Analysis

3.2.2 Operations

There are three common operations that can be performed on HMM—learning, inference and scoring.

Learning is used to estimate the model parameters $\{\pi, A, B\} = \lambda$. It is desired to estimate the parameters such that $\prod_{j=1}^{M} P(x_j|\lambda)$ is maximized. Where $M \in \mathbb{N}$ is a number of sequences used in learning. $x_j$ is a $j$-th sequence of observed symbols. In other words, it is desired to maximize the probability that given sequences of observed symbols were generated by the model.

Inference returns the most likely sequence of hidden states, given a sequence of observed symbols.

Given a sequence of observed symbols $x_1, x_2, \ldots, x_T$, the probability that the sequence was generated by HMM with parameters $\lambda$ is called a score.

3.2.3 Learning

**Initial transition function** The initial probability vector $\pi$ can be viewed as a probability function $\pi_i$, assigning a probability to every hidden state $S_i \in S$. This is a function of discrete random variable that defines discrete probability distribution. As such, MLE technique (discussed in chapter 2) can be used to estimate the distribution parameters.

**Transition function** Every row of $A$ defines a probability function $a_{ij}$. These functions are called transition functions. The transition functions define discrete probability distributions. Parameters of every such distribution can be estimated using MLE.

**Emission function** The observation probability functions $b_i(o)$ are functions of either discrete or continuous random variable that takes on values in set of observed symbols $X$. For purposes of this thesis, it is assumed that continuous variables follow a Gaussian or Gaussian mixture distribution.

If values of the set of observed symbols $X$ are discrete, then the emissions define discrete distributions and the parameters can be estimated through
3.2. Hidden Markov model

MLE. If the values of $X$ are continuous, then the emissions define continuous probability distributions. As discussed earlier, parameters of Gaussian distribution can be estimated using MLE. Parameters of Gaussian mixture distribution can be estimated through EM algorithm.

**Baum-Welch training** Baum-Welch training is an alternative learning algorithm that is not based on the MLE approach. It does not require the sequences of hidden states for learning. This algorithm will not be covered in this thesis, but its existence should be noted in case you are a curious reader [4].

3.2.4 Inference

Given a sequence of observed symbols, the most likely associated sequence of hidden states is calculated using the Viterbi algorithm. Let’s define $\alpha_t(i)$ to be the maximum probability of sequence of hidden states and observed symbols up to time point $t$:

$$\alpha_t(i) = \max_{y_1, y_2, \ldots, y_{t-1}} P(y_1, y_2, \ldots, y_{t-1}, y_t = S_i, x_1, x_2, \ldots, x_t | \lambda), \quad (3.1)$$

where $\lambda$ are parameters of HMM model.

**Initialization** At time point $t = 1$, the probability of being at hidden state $S_i$ is simply the initial probability of being at that state and a probability of being at observed state $x_1$ given hidden state $S_i$. Formally:

$$\alpha_1(i) = \pi_i b_i(x_1).$$

**Recursion** Taking into account the structure of HMM, the equation 3.1 can be rewritten recursively as

$$\alpha_t(j) = \max_i \alpha_{t-1}(i) a_{ij} b_j(x_t). \quad (3.2)$$

Hidden state at particular time point $t$ is determined by

$$s_t = S_{\arg\max_i \alpha_t(i)}.$$

**Complexity** At each time point $t$ and hidden state index $j$, the equation 3.2 loops through every hidden state index $i$. There are $N$ hidden states. In addition, the recursion continues for every time point $t$. There are $T$ time points. Thus, the overall complexity is $N^2 T$. 

17
3. Analysis

3.2.5 Scoring

In previous section, I have defined $\alpha_t(i)$ to be a probability of a sequence of hidden states up to time point $t$, where $y_t = S_i$, and a sequence of observed symbols up to time point $t$, given model parameters.

In order to compute the probability that sequence of observed symbols was generated by the HMM (score), take a look how $\alpha_T(i)$ is defined:

$$
\alpha_T(i) = \max_{y_1,y_2,\ldots,y_{T-1}} P(y_1,y_2,\ldots,y_{T-1},y_T = S_i, x_1,x_2,\ldots,x_T|\lambda).
$$

The score is then

$$
\sum_{i=1}^{N} \alpha_T(i).
$$

3.2.6 Example

Suppose an extension of an example mentioned in Markov chain chapter. i.e. there are three hidden states of weather: $S_1 =$ sunny, $S_2 =$ rainy and $S_3 =$ cloudy. Let’s introduce a continuous observed random variable describing humidity level. It is easy to imagine that on sunny day, the humidity will be typically lower than on a rainy day. This observable variable can help in predicting future weather. The example is depicted in figure 3.3.

3.2.7 Limitations

There is only one observed variable, which may not be sufficient for successful prediction in real-life applications. One could increase the dimensionality of the observed variable. That would, however, lead to a problem called curse of dimensionality. As we would increase the dimensionality of observed variable, the learning time would increase greatly up to the point where it would no longer be tractable. More training data would be necessary too. Dynamic naive Bayesian classifier addresses this issue.

3.3 Bayesian network

Bayesian network is a probabilistic graphical model consisting of nodes and edges. The model is a directed acyclic graph. Every node describes a random variable which can be either discrete or continuous. Edges describe a conditional dependency among the nodes (variables).

As an example, consider a Bayesian network in figure 3.4. In the example, each node represents a discrete random variable with two possible states—true and false. It can be interpreted that dehydration can cause red skin, but it can also influence presence of cancer. In addition, red skin may signal a lack
3.4 Naive Bayesian classifier

Naive Bayesian classifier is a classifier, where features (random variables) are assumed to be conditionally independent of each other (given class label variable).

Imagine we would like to classify a class $C_k$ given feature vector $x$:

$$P(C_k|x_1, x_2, \ldots, x_n).$$

Using Bayes’ theorem, the conditional probability can be rewritten as

$$P(C_k|x) = \frac{P(C_k, x)}{P(x)} = \frac{P(C_k)P(x|C_k)}{P(x)}.$$
### 3. Analysis

- Dehydration
- Cancer
- Red skin

#### Figure 3.4: Visualization of a Bayesian network with three states.

#### Table 3.1: CPT for dehydration.

<table>
<thead>
<tr>
<th>Dehydration</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>0.1</td>
<td>0.9</td>
</tr>
</tbody>
</table>

#### Table 3.2: CPT for cancer.

<table>
<thead>
<tr>
<th>Input</th>
<th>Cancer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dehydration</td>
<td>True</td>
</tr>
<tr>
<td></td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>0.1</td>
</tr>
<tr>
<td>False</td>
<td>0.01</td>
</tr>
</tbody>
</table>

#### Table 3.3: CPT for red skin.

<table>
<thead>
<tr>
<th>Input</th>
<th>Red Skin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancer</td>
<td>Dehydration</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td></td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>0.4</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

Table 3.3: CPT for red skin.
3.5 Dynamic Bayesian network

Dynamic Bayesian network is a Bayesian network which takes time into account. At each time point $t$, the values of random variables can be computed based on the values of associated random variables in time $t - 1$.

Consider figure 3.5 as an example. In the example, there is an airplane flight taking place. It can be interpreted that the velocity of an airplane at time $t$ conditionally depends on the velocity at time $t - 1$ as well as on the wind speed at time $t - 1$. Remaining gas at time $t$ conditionally depends on the remaining gas at $t - 1$ and it is also influenced by the altitude and the velocity at current time $t$.

3.6 Dynamic naive Bayesian classifier

3.6.1 Introduction

Dynamic naive Bayesian classifier (DNBC) can be considered as an extension of hidden Markov model. The difference is in the number of observed variables. HMM defines only a single observed variable, whereas DNBC supports multiple observed variables.
3. Analysis

![Diagram of a dynamic naive Bayesian classifier with two observed variables](image)

**Figure 3.6**: Visualization of a dynamic naive Bayesian classifier with two observed variables. At each time point, there is a hidden state $y_t$ and two observed states: $x^1_t$ and $x^2_t$.

DNBC is dynamic, because it classifies sequences with variables at every time $t$. It is called naive, because the output variables are assumed to be conditionally independent of each other. Figure 3.6 demonstrates a walk through a model with 2 observed variables.

### 3.6.2 Description

DNBC is defined by

- a number of observed variables $O$,
- a set of hidden states $S = \{S_1, S_2, \ldots, S_N\}$,
- sets of observed symbols $X^j, j \in [1, O]$
- a transition matrix $A \in \mathbb{R}^{N \times N}$,
- observation probability functions $B^j = \{b^j_1(o), b^j_2(o), \ldots, b^j_N(o)\}, j \in [1, O]$,
- an initial probability vector $\pi \in \mathbb{R}^N$.

The values of particular set of observed symbols $X^j$ can be either discrete or continuous. The values of $S$ are assumed to be discrete. The observation probability function $b^j_i(o)$ describes a probability $P(x^j_t = o | y_t = S_i)$, where $o \in X^j$. $b^j_i(o)$ is called an emission function.

Imagine a walk through DNBC graph. At time point $t$, there is a hidden state $y_t$ and $O$ observed symbols $x^j_t$, where $j \in [1, O]$. Value of each observed symbol conditionally depends on the value of the hidden state. The dependency is described by the emission function $b^j_i(o)$.

An important property of DNBC is that the observed variables are assumed to be independent of each other. Therefore, DNBC does not define any conditional dependency function between two observed variables.

If $t \neq 1$ and $T > 1$, there is a hidden state $y_t$ that conditionally depends on the value of previous hidden state $y_{t-1}$. The dependency is described by transition $a_{ij}$. If $t = 1$, there is a hidden state $y_1$ that conditionally depends on the initial transition $\pi_i$. 

22
3.6.3 Notation

Let \( \hat{x}_t \) be the observed variables at time \( t \).

\[
\hat{x}_t = x^1_t, x^2_t, \ldots, x^O_t.
\]

It must hold that the number of observed variables at every time \( t \) is the same.

Let \( x_t \) be a sequence of observations up to time point \( t \).

\[
x_t = \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_t.
\]

For simplification, let’s define \( x \) to be all the observations of an input sequence.

\[
x = x^T.
\]

3.6.4 Operations

Similarly to HMM, there are three common operations that can be performed on DNBC—learning, inference and scoring.

Learning is used to estimate the model parameters \( \{\pi, A, B^{1..O}\} = \lambda \). It is desired to estimate the parameters such that \( \prod_{j=1}^{M} P(x^j|\lambda) \) is maximized. Where \( M \in \mathbb{N} \) is a number of sequences to be learned. \( x^j \) is a \( j \)-th sequence of observed symbols. In other words, it is desired to maximize the product of probabilities that the given sequence of observed symbols was generated by the given model.

Inference returns the most likely sequence of hidden states, given a sequence of observations.

Given a sequence of observations \( x \), the probability that it was generated by DNBC with parameters \( \lambda \) is called a score.

3.6.5 Learning

The learning approach is similar to the one described in the case of HMM. The only difference is that there are multiple emission functions \( b_j^i(o) \) where \( j \in [1, O] \) and \( O \) is the number of observed variables.

3.6.6 Inference

Viterbi algorithm can be applied in the similar way as in the case of HMM. It is extended to support multiple observed variables. Let’s define \( a_t(i) \) to be the maximum probability of sequence of hidden states and sequence of observations up to time point \( t \):

\[
a_t(i) = \max_{y_1, y_2, \ldots, y_{t-1}} P(y_1, y_2, \ldots, y_{t-1}, y_t = S_i | x_t | \lambda). \quad (3.3)
\]
### Analysis

**Initialization** At time point \( t = 1 \), the probability of being at hidden state \( S_i \) is equal to the initial probability of being at that state and probabilities of being at observed symbol \( x_1^j \) given hidden state \( S_i \). In other words:

\[
\alpha_1(i) = \pi_i \prod_{j=1}^{O} b_i^j(x_1^j).
\]

**Recursion** Taking into account the structure of DNBC, the equation \( 3.3 \) can be rewritten recursively as

\[
\alpha_t(j) = \max_i \alpha_{t-1}(i)a_{ij} \prod_{k=1}^{O} b_j^k(x_t^k).
\]

(3.4)

Hidden state at particular time point \( t \) is determined by

\[
s_t = S_{\arg \max_i \alpha_t(i)}.
\]

**Complexity** At each time point \( t \) and hidden state index \( j \), the equation \( 3.4 \) loops through every hidden state index \( i \) and every observed variable index \( k \). There are \( N \) hidden states and \( O \) observed variables. In addition, the recursion continues for every time point \( t \). There are \( T \) time points. Thus, the overall time complexity is \( N(N + O)T \).

#### 3.6.7 Scoring

Scoring is performed the same way as in case of HMM. That is, the score can be calculated as

\[
\sum_{i=1}^{N} \alpha_T(i).
\]

#### 3.6.8 Example

Consider a problem of weather prediction. Let’s define a set of hidden states to be \( S = \{S_1 = \text{sunny}, S_2 = \text{rainy}, S_3 = \text{cloudy}\} \). Let’s consider the following observed variables: temperature, humidity and windiness. At every time \( t \), the hidden state conditionally depends on the hidden state at time \( t-1 \). The three observed variables at time \( t \) conditionally depend on the hidden state \( t \). These variables can help in predicting the hidden state.

#### 3.6.9 Limitations

The observed variables are assumed to be conditionally independent of each other. While this assumption enables the use of algorithms with relatively low time complexity and less data, it also limits the ability to describe more
3.6. Dynamic naive Bayesian classifier

complicated relations among random variables. For example, the observed variables may be correlated in reality.
There are no existing DNBC libraries written in Scala language. Although there exists a library for Bayesian networks [7], which is a super set of DNBC, it does not support computations on top of Apache Spark. Bayesian networks use different learning and inference algorithms than DNBC. Extending this library would thus be above scope of this thesis.

I have implemented a dynamic naive Bayesian classifier in Scala programming language that runs on top of Apache Spark. I named this software \textit{dnbc-scala}.

### 4.1 Apache Spark

Apache Spark is an engine for cluster-computing. It provides application programming interface (API) that developers can use to execute computations on multiple nodes—processors or machines in a cluster. Spark is also fault-tolerant, meaning that if a computation fails at a given node, another node will take over.

When executing a program on top of Spark, it runs as a \textit{driver}. The driver passes execution of parallel operations such as \textit{map} or \textit{reduce} to Spark. These operations are then processed on multiple nodes in the cluster [12].

Map and reduce are fundamental operations in functional programming. The map operation takes a set of input data and converts it into another set. For example, one can transform a set of strings into a set of their corresponding lengths. Reduce takes a set of input data and combines it together. For example, one can take a set of numbers and add them together.

The map operation assumes that the input data set can be independently converted into another set. This enables the operation to run in parallel. The reduce operation assumes that the input data can be combined in any order. Thus the operation can be also performed in parallel fashion.

Apache Spark is implemented in Scala programming language. The language combines functional and object-oriented programming. Scala code com-
4. Implementation

preprocessing
core
test-utils
core-test

Figure 4.1: Dependencies among modules in 

piles into Java virtual machine (JVM) bytecode. This enables one to incorporate existing Java libraries into a Scala project.

4.2 Project structure

The project is divided into 5 modules:

- **core**—main module containing DNBC interface, helper class for loading data set into required format and a class for measuring performance of DNBC on a data set,
- **core-test**—contains tests of **core** module,
- **performance**—provides command line interface for generating a parametric data set and measuring DNBC performance on it,
- **preprocessing**—transforms a predefined data set into data sets with different parameters,
- **test-utils**—provides commonly used functions in tests or in performance measurement.

Figure [4.1] provides overview of the project structure and module dependencies.

4.3 Usage

DynamicNaiveBayesianClassifier object  DynamicNaiveBayesianClassifier object in **core** module serves as a factory for already learned class DynamicNaiveBayesianClassifier.
4.3. Usage

object DynamicNaiveBayesianClassifier {
  def mle(sc: SparkContext,
    sequences: Iterable[Seq[State]],
    continuousVariableHints: Option[List[Int]] = Option.empty)
    : DynamicNaiveBayesianClassifier
}

The function \textit{mle} returns a learned model. \textit{sequences} are sequences of \textit{States} used for learning. Optional argument \textit{continuousVariableHints} can be provided to set the number of components in Gaussian mixtures. The implementation does not try to estimate this number. If the parameter is not provided, the continuous variables are expected to be normally distributed.

The model parameters (\textit{Edges}) are learned in every sequence of states. After processing of all input sequences, learning is finalized by calling a function \textit{learnFinalize} on each edge. The input sequences are used altogether to do the actual learning. The \textit{learnFinalize} functions are executed in parallel since their execution order is not important and they do not depend on each other.

\textbf{Learned DNBC class} The following class represents a DNBC model with known parameters.

\begin{verbatim}
class DynamicNaiveBayesianClassifier(
  initialEdge: LearnedDiscreteEdge,
  transitions: Map[String, LearnedDiscreteEdge],
  discreteEmissions: List[Map[String, LearnedDiscreteEdge]],
  continuousEmissions: List[Map[String, LearnedContinuousEdge]]) {
  def inferMostLikelyHiddenStates(observedStates: Seq[ObservedState]): List[String]
  def score(observedStates: Seq[ObservedState]): Double
}
\end{verbatim}

The \textit{inferMostLikelyHiddenStates} function returns the most likely sequence of hidden states given a sequence of observed states. The function uses Viterbi algorithm. The following code is a function for Viterbi initialization.

\begin{verbatim}
private def viterbiInitialize(observedStates: Seq[ObservedState]): Map[String,Double] = {
  var vcur = Map.empty[String,Double]
  for (hiddenState <- transitions.keys) {
    var emissionsSum = 0.0
    emissionsSum += discreteEmissions.zipWithIndex.map(z =>
      Math.log(z._1(hiddenState).
          probability(observedStates.head.DiscreteVariables(z._2))).sum
    emissionsSum += continuousEmissions.zipWithIndex.map(z =>
      Math.log(z._1(hiddenState).
          probability(observedStates.head.ContinuousVariables(z._2))).sum
    vcur += hiddenState -> emissionsSum
  }
  vcur
}
\end{verbatim}
4. Implementation

```scala
vcur += (hiddenState -> (emissionsSum +
    Math.log(initialEdge.probability(hiddenState))))
}
```

The `score` function returns a log-probability that given sequence of observed states was generated by the DNBC. It is internally using Viterbi algorithm to compute $\alpha_T(i)$.

The Viterbi algorithm is not implemented in parallel. The reason is that the algorithm proceeds by time points, and each time point depends on the previous one. At every time point, the parallelization is not worth it, since there is only a bunch of additions taking place. However, parallelization over hidden states might be useful if the number of hidden states is huge. This optimization can be considered in a subsequent development of this project.

**State** An observed state at given time point consists of observations of all the discrete and continuous variables. This is represented by `ObservedState` class.

```scala
class ObservedState(discreteVariables: List[String],
    continuousVariables: List[Double]) {
    def DiscreteVariables: List[String] = discreteVariables
    def ContinuousVariables: List[Double] = continuousVariables
}
```

A state at given time point consists of a hidden state and an `ObservedState`. Throughout the project, the hidden states are of `String` type.

```scala
class State(hiddenState: String, observedState: ObservedState) {
    def HiddenState: String = hiddenState
    def ObservedState: ObservedState = observedState
}
```

**Edge** An edge represents a particular random variable, parameters of which are to be learned.

```scala
trait Edge[T] {
    def learn(occurrence: T): Unit
    def learnFinalize(): LearnedEdge[T]
}
```

The `learn` function notifies an edge that `occurrence` occurred. User calls `learnFinalize` function to estimate the variable parameters based on previous occurrences.
There are two kinds of edges: \textit{DiscreteEdge} and \textit{ContinuousEdge}. As expected, \textit{DiscreteEdge} represents a discrete random variable, whereas \textit{ContinuousEdge} represents a continuous random variable.

To estimate the parameters of continuous random variables using MLE, I am using jMEF Java library. Previously I used Spark’s MLlib library for estimating parameters of a Gaussian mixture model (GMM). The problem with this approach was that it can only estimate the parameters by making a computation across multiple nodes in a cluster. This is not a desired behavior since there are many estimations of small data that should run in parallel instead. Therefore, I ended up using jMEF library which estimates the parameters in sequential fashion.

\textbf{LearnedEdge} A \textit{LearnedEdge} represents a random variable which parameters have been already estimated. It has a \textit{probability} function that returns a probability (density) of the state.

\textbf{RandomEdge} A \textit{RandomEdge} generates random value given random variable parameters. This is useful for performance measurement, where data sets are randomly generated.

### 4.4 Underflow

Since the Viterbi algorithm is computing products of probabilities, the resulting value gets small in few iterations. To avoid underflow, one can compute a log-probability instead, using the following trick:

\[
\log(abc) = \log(a) + \log(b) + \log(c).
\]

Now instead of multiplying tiny numbers together, one can use addition.

### 4.5 Data set generation

The \textit{test-utils} module contains a function \textit{generatePerformanceDataSet}. This generates a random data set, given parameters. The data set can be used for measuring learning and testing time. The supported parameters are:

1. the length of sequences,
2. the number of training sequences,
3. the number of testing sequences,
4. the number of hidden states,
5. the number of discrete observed variables,
4. Implementation

6. the number of continuous observed variables,

7. the maximum number of components per mixture,

8. the number of transitions per hidden state.

The initial edge probabilities are generated by choosing random number from a Gaussian distribution. The numbers are then normalized to ensure their sum is equal to one. Every hidden state’s transition destination is chosen randomly from an uniform distribution. The probabilities of transitions are then set the same way as in case of initial edge. Same approach is used for discrete emissions.

For continuous emissions, the parameters of GMM are chosen randomly in uniform fashion in range of these parameters:

\[ \mu \in [-10, 10], \sigma^2 \in [1, 4]. \]

The weights in a RandomContinuousEdge are all equal.

4.6 Source code

The implementation is available on GitHub [13] or via the provided CD that is part of this thesis.
Experiments

In this section, I conduct experiments of two kinds. First, I focus on how well DNBC learns a realistic data set in terms of average success rate. Then I measure scalability potential and learning time in terms of speed up on randomly generated data sets.

5.1 Realistic data set

5.1.1 Data set description

I am using a Toy Robot data set as a base data set. Imagine a robot wandering through a maze. There are 12 hidden states. Every hidden state is a 2D coordinate. Each hidden state has an associated color. The color is a discrete observed variable that can take on 4 states: red, green, blue and yellow. The initial state is chosen randomly.

The data set contains of 200 training sequences and 200 test sequences. Each sequence consists of 200 time points. 10% of the data set observed states at time points are chosen at random. This simulates robot’s occasional failure to detect correct color.

Measurement For each realistic data set, an average success rate (ASR) is measured. This average is the average percentage of hidden states inferred correctly in the testing phase.

5.1.2 Experiment design

Experiments were divided into 5 scenarios. Each scenario differs in type of observed variable(s). The purpose of these experiments is to show that my implementation yields reasonable results.
5. Experiments

Discrete variable In this scenario, the Toy Robot data set contains a single observed discrete variable.

Continuous variable The Toy Robot data set is transformed to have a single continuous observed variable instead. Each color (discrete random variable) is replaced by a temperature. The idea is that the aforementioned 4 colors have different temperature distributions. These distributions, however, partially overlap.

Discrete and continuous variable A discrete observed variable is added to the continuous variable data set. This variable describes the quadrant of given 2D coordinate. It is assumed that these two variables are independent.

Gaussian mixture Point of this test is to check that providing a variable hint describing the number of mixture components improves the ASR. If the hint is not provided, the mixture would be represented by a Gaussian distribution that would interfere with the other Gaussian distribution. A data set with single continuous observed variable is generated. Hidden states can take on two values: true and false. If the hidden state is true, observed state is generated from a Gaussian distribution. If the hidden state is false, observed state is generated from a Gaussian mixture distribution. The problem is demonstrated in figure 5.2.

Scoring A score is computed for a sequence of observed states that comes from a data set a DNBC was trained on. Another score is computed for a sequence of observed states that was generated randomly. Since the score is a log-probability, the values are negative. The following holds:

\[ \text{scoreReal} \times 1.5 > \text{scoreRandom}. \]
5.1. Realistic data set

Figure 5.2: Visualization of the Gaussian mixture data set. The solid curves represent a distribution of the observed continuous variable given hidden state is false. If the hidden state is true, the variable follows a Gaussian distribution as visualized by the dotted curve.

<table>
<thead>
<tr>
<th>data set</th>
<th>ASR [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>discrete</td>
<td>65</td>
</tr>
<tr>
<td>continuous</td>
<td>42</td>
</tr>
<tr>
<td>combined</td>
<td>76</td>
</tr>
<tr>
<td>Gaussian mixture (without hint)</td>
<td>96</td>
</tr>
<tr>
<td>Gaussian mixture (with hint)</td>
<td>99</td>
</tr>
</tbody>
</table>

Table 5.1: Results on realistic data sets. There are two main reasons for relatively low success rates on the first two data sets: only 90% of observed symbols are accurate and there are often multiple transitions to hidden states with the same observed symbol.
5. Experiments

<table>
<thead>
<tr>
<th>CPU</th>
<th>2× 8-core Intel Xeon E5-2650 v2 2.6 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>96 GB</td>
</tr>
<tr>
<td>disk</td>
<td>2× 500 GB HDD WD Velociraptor 10k SATA</td>
</tr>
</tbody>
</table>

Table 5.2: Machine specification [3]

<table>
<thead>
<tr>
<th>number of processors</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>length of sequences</td>
<td>200</td>
</tr>
<tr>
<td>number of training sequences</td>
<td>1000</td>
</tr>
<tr>
<td>number of testing sequences</td>
<td>200</td>
</tr>
<tr>
<td>number of hidden states</td>
<td>10</td>
</tr>
<tr>
<td>number of discrete observed variables</td>
<td>5</td>
</tr>
<tr>
<td>number of continuous observed variables</td>
<td>5</td>
</tr>
<tr>
<td>max number of components per mixture</td>
<td>3</td>
</tr>
<tr>
<td>transitions per hidden state</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 5.3: Base data set generation and evaluation parameters.

5.2 Random data set

In this section, I focus on learning time given a generated parametric data set. The measurements were made on a machine specified in table 5.2. Every job was assigned 15 GB RAM and 10 GB disk space. Actual resource usage always stayed within the limits.

The base parameters used for data set generations and performance measurement are described in table 5.3.

Local scalability Goal of this measurement is to see how well the implementation scales as I increase the number of processors. Only a single machine is used. The data sets had 30 discrete and 30 continuous observed variables. Figure 5.3 shows results of this measurement. Each configuration was measured 4 times to enable better understanding of the data.

As it can be seen, the software scales relatively well up to about 8 workers. The speed up is not dramatic, however. In fact, the majority of time is spent reading the data set data. This is not a parallel operation. Reading data is the bottleneck.

To focus on scalability of learning itself, figure 5.4 shows learning time of all continuous emissions, without the data loading. As it can be seen, this segment scales even better.

Cluster scalability In this experiment, I set up 8 independent worker nodes in a cluster. Every worker is assigned 15 GB RAM and 10 GB disk space. The goal is to measure scalability as the number of processors per node increases.
The data sets had 100 discrete and 100 continuous observed variables. As it can be seen on figure 5.5, the continuous emissions learning time is about 45% lower when using $8 \times 4$ processors in comparison to using 8 processors.

**Training sequences** Figure 5.6 shows approximately linear time increase as number of training sequences grows.

**Hidden states** Interestingly, as I increase the number of hidden states, the learning time does not change. This can be see in figure 5.7.

**Discrete variables** As the number of observed discrete variables increases, the learning times increases approximately linearly. Take a look at figure 5.8.
5. Experiments

Figure 5.5: Continuous emissions speed up ratio per number of processors per worker.

Figure 5.6: Learning time per number of training sequences.

Figure 5.7: Learning time per number of hidden states.
5.3 Summary

I have experimentally proved correctness of my implementation by measuring ASR on a realistic data sets. Scalability was demonstrated not only locally—on a single machine, but also in a computational cluster. As it can be seen in the aforementioned plots, the speed up is higher when using only a single machine. The communication between multiple machines takes extra time. The cluster mode may be more suitable on data sets with large number of observed variables or many transitions per hidden state.

Continuous variables Similar phenomenon can be seen when the number of observed continuous variables increases. The plot looks very similar to the one where number of discrete variables is being increased.
Conclusion

I have described the theory behind dynamic naive Bayesian classifier (DNBC) and other models it is based on. Since no known suitable implementation currently exists in Scala, I have decided to implement DNBC by myself. I have demonstrated correctness of my implementation through realistic data sets. Furthermore, the software shows promising results in terms of speed up when used in a system with multiple processors. It has been shown that the learning time needed can be lowered by a half.

Future improvement could be to implement an alternative learning algorithm: Baum-Welch training, and compare it to existing maximum likelihood estimation (MLE) based learning.
Bibliography


43
Bibliography


List of abbreviations

ASR  average success rate
API  application programming interface
CPT  conditional probability table
DNBC  dynamic naive Bayesian classifier
EM  expectation-maximization
GMM  Gaussian mixture model
HMM  hidden Markov model
i.i.d.  independent, identically distributed
JVM  Java virtual machine
PDF  probability density function
MC  Markov chain
The content of the enclosed CD

- readme.txt ......................... brief description of the CD content
- src
  - impl ............................ implementation source code
  - thesis........................... thesis source code in $\LaTeX$ format
- thesis.pdf ........................ thesis in PDF format