UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

DZNE

Generic Generation of Hidden Markov Models and an Application to Medical Data

Bachelorarbeit

zur Erlangung des Grades Bachelor of Science (B.Sc.)
im Studiengang Informatik

vorgelegt von
Richard Fechner

Erstgutachter:	Prof. Dr. J. Jürjens
	Fachbereich 4, Institut für Softwaretechnik
Zweitgutachter:	Dr. J. Dörpinghaus
	Fachbereich 3, Mathematisches Institut

Koblenz, im September 2022

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Contents

1 Introduction

1.1 Motivation

At the heart of the field of Machine Learning (ML) lies the idea to infer the unknown solely from observation. Every environment emits signals to us, may it be rays of light, which we perceive through our eyes, sound waves, which we can hear with our ears, or other signals we can measure with our instruments. Measuring these signals for consecutive discrete timesteps presents us with a so-called multivariate time-series - or to put it in other words - a series of multiple observable variables over a period of time. Inferring knowledge about a state-sequence of an unknown variable from these sequences of observations is what this thesis is primarily concerned with. The novelty presented in this thesis is the capability of choosing, which unknown quantity to infer from observation. The model is not only able to infer knowledge about quantity A by observing quantity B but allows for the exploration in the opposite direction, namely reasoning about B from observing A (see Figure 1). Furthermore, this thesis extends this basic principle to a more general case, in which the model allows for maximum flexibility and generality by being able to choose any constellation of different observable variables to infer knowledge about any other observable variable. This is realized within the widely used probabilistic framework of Hidden Markov Models, which this thesis will explore and augment, before applying the model to real world medical data.

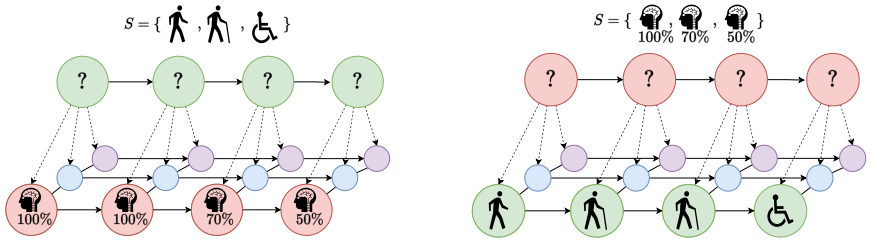


Figure 1: Conceptual depiction of the flexibility enabled by the proposed model. The predicted variable as well as the means by which to make a prediction are chosen freely by the user.

Hidden Markov Models (HMMs) have been successfully applied to the processing of possibly noisy continuous signals in the case of speech or gesture recognition [7, 14, 15], as well as to the processing of sequences of discrete signals like text categorization [10]. In the context of medicine, they have been incorporated into forecast models, improving estimates about patient mortality [17]. In this thesis, I'll focus on multivariate time-series of categorically distributed data. I'll provide a prediction pipeline in which a user can input their data paired with a configuration file, enabling them

to construct, validate and query a fully parameterized HMM-based model. Specifically, I'll propose a rather new technique that includes constructing multiple HMMs - one for each variable sequence of the multivariate sequence - to predict another observable variable. The questions I would like to ask are, whether the newly proposed technique can perform to a degree of satisfaction when tested on real world application data, what the strengths and weaknesses of the model are, and how the approach could be augmented to improve results.

1.2 Aim and Structure of this Thesis

The aim of this thesis is to a) provide a theoretical and practical framework for multivariate time-series inference based on HMMs and b) apply the proposed prediction pipeline (see Figure 2) to real world data.

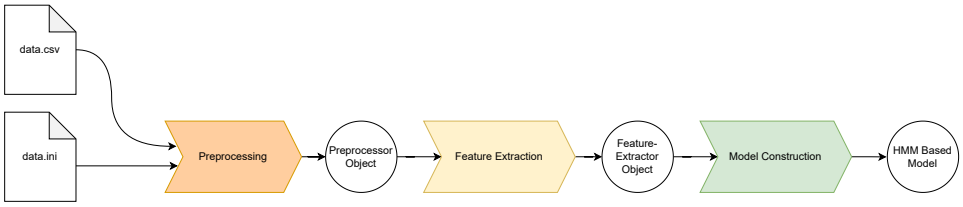


Figure 2: Concept of the presented prediction pipeline, allowing for HMM-based model construction and model query.

To pave the way for the latter parts of this thesis, I'm first going to cover the most fundamental basics, including Graph Theory, the basic theory of Hidden Markov Models, big-O notation and Machine Learning metrics in Section 2. After having laid out the most fundamental theory, I'll continue in Section 3, introducing notation and giving an in-depth overview of the inner workings of the prediction pipeline as well as model prediction capabilities. Going further, in Section 4 I'll apply the previously presented techniques on a - for exemplary purpose synthesized - dataset, before shifting the attention to the application on real world medical data supplied by the DZNE (German Center for neurodegenerative diseases). In the last Section, I'll give an interpretation of the results and discuss the performance of the model, before closing with an outlook on possible augmentations and improvements of the model.

2 Foundations

2.1 Graph Theory

2.1.1 Graphs

A Graph is an abstract mathematical object used to model relationships between objects. In its simplest form, an **undirected Graph** can be used to model relationships between certain entities. The formal definition is as follows.

Definition 1 *An **undirected Graph** $G = (V, E)$ is a tuple containing a set of vertices $V = \{v_1, v_2, ..., v_N\}$, sometimes also called nodes, and a set of edges $E \subseteq V \times V$ connecting the vertices with each other. Additionally, $(v_a, v_b) \in E \iff (v_b, v_a) \in E$.*

Definition 2 *A **directed Graph** $G = (V, E)$ is a tuple of the set of vertices and edges. The direction of the edge e_i is inferred from the order of the tuple by which e_i is defined by.*

To give an example, the edge $e_k = (v_a, v_b)$ connects the vertex v_a to vertex v_b. With the introduced notion of direction, one can model more complex dynamics, like for example different states connected by state transitions.

Finally, one can augment the edges between the vertices, by assigning weights to them. A weight function $w : E \to \mathbb{R}$ is introduced, which maps an edge e to their corresponding edge-weight $w(e)$. A weight $w(e)$ corresponding to an edge $e = (v_a, v_b)$ can be interpreted as the cost of traveling from vertex v_a to vertex v_b. Alternatively, we can interpret the weight of a transition (edge) between two states (vertices) as the probability of transitioning between the two. The latter interpretation will be very useful to describe a system of states and the transitions between them, where the probability of moving from one state to another in a single timestep is denoted by the weight of the edge connecting the two states.

2.1.2 Matrix Representation of a Graph

An undirected, unweighted Graph $G = (V, E)$ can be represented G as a symmetric $|V| \times |V|$ matrix $\mathcal{A}$, where the elements of matrix a_{ij} are marked as

$$a_{ij} = \begin{cases} 1 & (v_i, v_j) \in E \\ 0 & else \end{cases}$$

such that the entry a_{ij} in the matrix indicates, whether the vertices v_i and v_j are connected to one another.

Extending this idea to directed, weighted Graphs comes naturally. Suppose we have a directed, weighted Graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$ respecting stochastic constraints, namely that the sum of all weights for out-going edges is to equal 1, for all vertices

$$\sum w(e_i) = 1, e_i \in \{e_k \in E | e_k = (u, v)\} \qquad \forall u \in V$$

and that all edge-weights must be positive.

$$w(e) \geq 0, \qquad \forall e \in E$$

This allows G to be represented as a **row-stochastic matrix** $\mathcal{A}$, where a_{ij}

$$a_{ij} = \begin{cases} w(e_k) & e_k = (v_i, v_j) \in E \\ 0 & else \end{cases}$$

denotes the probability of transitioning (weight of the connecting edge) from state (vertex) i to state (vertex) j in a single timestep.

2.2 Markov Chains

A first order **Markov Chain** is a stochastic process enabling us to model states and their state-transitions.

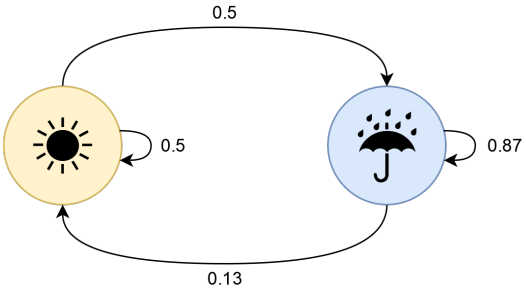
Definition 3 *A Markov Chain is defined by the set of N states $S = \{S_1, ..., S_N\}$ and a $N \times N$ row-stochastic transition matrix $\mathcal{A}$ where a_{ij} defines the probability of transitioning from state S_i to state S_j in a single timestep.*

We will be concerned with first order Markov Chains, which satisfy the so called **Markov Property**.

Definition 4 *The **Markov Property***

$$P[q_t = S_i \mid q_{t-1} = S_a] = P[q_t = S_i \mid q_{t-1} = S_a, ..., q_{t-k} = S_x]$$

states that the probability for state transition is solely dependant on the previous state, regardless of any other states visited before that. Here, q_t denotes the state at timestep t.



$$\mathcal{A} = \begin{bmatrix} 0.5 & 0.5 \\ 0.13 & 0.87 \end{bmatrix}$$

$$\pi = \begin{bmatrix} 0.21 & 0.79 \end{bmatrix}^T$$

Figure 3: A Markov Chain Model of the weather consisting of two states, sun and rain. The transition probabilities are displayed in the transition matrix $\mathcal{A}$.

2.3 An Introductory Example

Suppose, that we want to model the weather as a Markov Chain with exactly two states, S (for sunny) and R (for rainy). By observing the actual weather for some time, we get a rough estimate of what the transition probabilities between the states are, thus obtaining the transition probability matrix $\mathcal{A}$. In addition, we are now able to compute the stationary distribution π from our transition matrix. Here, π_{S_i} denotes the probability of starting in state S_i. Later, we will call π the initial state distribution. Now, we use our model (see Figure 3), let us call it $\lambda = (\mathcal{A}, \pi)$, to reason about the probability of certain state-sequences. One question we could ask might be "*What is the probability of observing a sunny day, then another sunny day, finally a rainy day, out of all possible 3-day observations?*".

We can answer this question by querying our model. Since our model satisfies the **Markov Property**, we know that $P[\mathcal{O} = SSR \mid \lambda]$ the probability of observing the given observation sequence, given our model λ comes out to be the product of the probability of initially starting in state S (π_S), the transition probability from state S to itself (a_{11}) and the transition probability from state S to state R (a_{12}).

$$\begin{aligned} P[\mathcal{O} \mid \lambda] &= P[q_1 = S, q_2 = S, q_3 = R \mid \lambda] \\ &= P[q_1 = S \mid \lambda] \cdot P[q_2 = S \mid q_1 = S, \lambda] \cdot P[q_3 = R \mid q_2 = S, \lambda] \\ &= \pi_S \cdot a_{11} \cdot a_{12} \end{aligned}$$

This simple framework allows us to assign a probability to any observable sequence. The subsequent paragraph will show how to extend the framework to the inference of probability for sequences of non-observable hidden states based on observations.

2.4 From Markov Chain Model to Hidden Markov Model

In the previous example, the states in our Markov Chain Model were directly observable. However, suppose that the states we would like to reason about are not directly observable and are hidden from us. This motivates the extension of a Markov Chain Model to a **Hidden Markov Model**, in which the actual states we would like to reason about are hidden and cannot be observed. We can only observe *emission-signals*, which are being emitted from the hidden states. The following definitions are taken from Rabiners' very influential tutorial on HMMs [16].

Definition 5 *A **Hidden Markov Model** $\lambda = (\mathcal{A}, \mathcal{B}, \pi)$ with the hidden states $S = \{S_1, ..., S_N\}$ and emission signals $V = \{V_1, ..., V_M\}$ is a stochastic process, defined by three model parameters*

1. *$N \times N$ row-stochastic **state transition matrix** $\mathcal{A}$, where the element a_{ij} denotes the state transition probability from state S_i to state S_j.*

   An alternative notation for denoting the state transition from q_{t-1} to q_t is given by $a_{q_{t-1}q_t}$.

2. *$N \times M$ row-stochastic **state emission matrix** $\mathcal{B}$ where the element b_{ij} denotes the probability $P[o_t = V_j | q_t = S_i]$ of observing emission signal $o_t = V_j$ at timestep t under the hidden state S_i.*

   An alternative notation for denoting the observation of signal o_t from the hidden state q_t is given by $b_{q_t}(o_t)$.

3. *$1 \times N$ **initial state distribution vector** π, where $\pi_i = P[q_1 = S_i]$.*

   An alternative notation for denoting the initial state probability for hidden state q_t is given by π_{q_t}.

Coming back to our simple weather model, let us transform the Markov Chain Model into a Hidden Markov Model. Assume, that we are unable to observe the actual state of the weather (maybe our window blinds are down). The only thing we can observe is the presence or absence of dogs barking, whilst they are being taken on a walk. Thus, we have two **hidden states**, namely R (for rainy) and S (for sunny), as well as two **emission signals**, namely B (for barking) and $\neg B$ (for not barking). For the sake of simplicity, let us assume that we were able to estimate the transition probabilities for the hidden states, as well as the emission probabilities for each

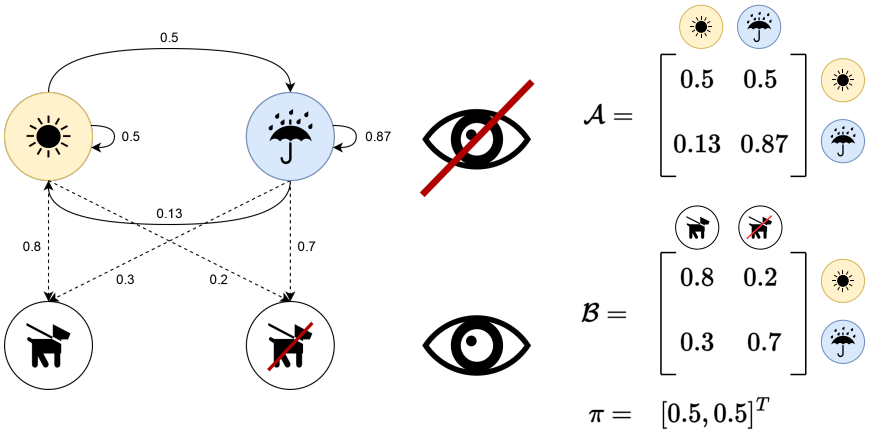


Figure 4: A Hidden Markov Model of the weather with two hidden states and two observable states.

hidden state, and noted them into our row-stochastic matrices $\mathcal{A}$ and $\mathcal{B}$ respectively. Furthermore, let us assume that the initial state distribution is given with $\pi_S = \pi_R = \frac{1}{2}$.

Having all model parameters $(\mathcal{A}, \mathcal{B}, \pi)$ fixed, we can reason about the hidden states solely from made observations. For example, we could have observed the sequence $\mathcal{O} = BBB\neg B\neg B$ and now ask ourselves what the most likely weather state sequence is.

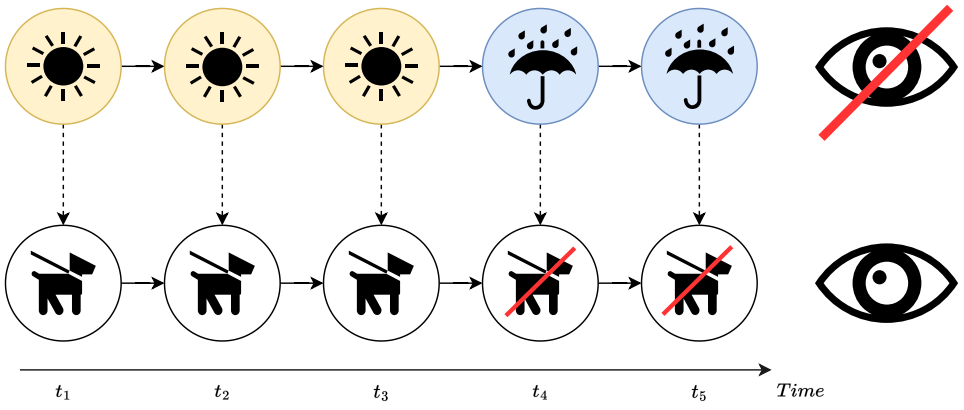


Figure 5: What is the most likely state sequence for our weather to have produced our observation sequence?

These types of questions, where we would like to infer a sequence of hidden states from an observation sequence are very common. In the medical context for example we might be interested in inferring a sequence of diag-

noses from the patients observed heartbeat and breathing frequency.

2.5 Three Basic Problems for HMMs

Generally, we are concerned with inferring a sequence of hidden states from a sequence of observed states. This is what's referred to as a **sequence-to-sequence** prediction. The Hidden Markov Model excels at this task, although we will see that the path towards a stable prediction poses some challenges and conceals certain pitfalls we need to avoid. Following Rabiners tutorial on HMMs [16] we are presented with three main problems we need to solve to be able to apply HMMs in practice.

Problem 1: Given an observation sequence $\mathcal{O} = o_1 o_2 ... o_T$, as well as a fully parameterized model $\lambda = (\mathcal{A}, \mathcal{B}, \pi)$, how do we calculate the probability $P[\mathcal{O}|\lambda]$ in an efficient manner?

Problem 2: Given an observation sequence $\mathcal{O} = o_1 o_2 ... o_T$ as well as a fully parameterized model $\lambda = (\mathcal{A}, \mathcal{B}, \pi)$, how do we find the state sequence $\mathcal{Q} = q_1 q_2 ... q_T$ best explaining the seen observation?

Problem 3: Given a model λ, how do we train the model, changing its parameters to maximize $P[\mathcal{O}|\lambda]$?

Solving the first problem enables us to compare different Hidden Markov Models. Given an observation sequence $\mathcal{O}$, we might prefer the model λ, which maximizes $P[\mathcal{O}|\lambda]$, the probability of observing the given sequence based on model parameters.

A solution to the second problem allows for an inference of a sequence of hidden states $\mathcal{Q}$ given an observation sequence $\mathcal{O}$. Since many different hidden state sequences might produce the given observation sequence, the problem becomes finding a $\mathcal{Q}$, which maximizes $P[\mathcal{O}|\mathcal{Q}, \lambda]$, the probability of the hidden state sequence $\mathcal{Q}$ emitting the observation sequence $\mathcal{O}$.

The third problem is to approximate the in practice **unknown** model parameters $\mathcal{A}, \mathcal{B}, \pi$ given an observation sequence $\mathcal{O}$. This can be interpreted as training an HMM on an observation sequence.

2.5.1 Solution to Problem 1

Problem 1 is the problem of evaluating the probability of observing an observation sequence $\mathcal{O}$ given a model λ. Solving this problem enables us to compare different models, thus letting us decide which model "best fits" our observation. This problem is best approached naively at first, without

having computational cost in mind. As we will see, the need for a smarter, less computationally intensive solution will arise along the way.

First, consider a fixed state sequence $Q = q_1...q_T$ of some states that might emit the observation sequence. The probability of this sequence arising from the model is given by the product of the probability of starting in the state q_1 which is given by π_{q_1} and the correct state transition probabilities.

$$P[Q|\lambda] = \pi_{q_1} \prod_{t=1}^{T-1} a_{q_t q_{t+1}} = \pi_{q_1} \cdot a_{q_1 q_2} ... \cdot a_{q_{T-1} q_T}.$$

Additionally, the probability of observing the observation sequence $\mathcal{O}$ from the state sequence Q is given by the product of the single emission probabilities for the individual observations under the hidden state.

$$P[\mathcal{O}|Q, \lambda] = \prod_{t=1}^{T} b_{q_t}(o_t) = b_{q_1}(o_1) \cdot ... \cdot b_{q_T}(o_T)$$

Finally, the probability for observing $\mathcal{O}$ under Q is $P[Q|\lambda] \cdot P[\mathcal{O}|Q, \lambda]$. Having computed this probability for one possible state sequence, all that is left is computing it again for every single possible state sequence of length T.

$$P[\mathcal{O}|\lambda] = \sum_{Q \in \mathcal{Q}} P[Q|\lambda] \cdot P[\mathcal{O}|Q, \lambda]$$

Where, $\mathcal{Q}$ is the set of all possible state sequences of length T. Although very declarative, this solution is practically useless since the computational effort required to solve for only one observation sequence increases exponentially with the length T of the sequence, as for every timestep there are up to N different state transitions to be made, resulting in a total of at worst N^T different candidates for Q which are to be evaluated. This is unfeasible for even a small number of hidden states N and a short sequence length.

To overcome this hurdle, we make use of a programming paradigm named "dynamic programming". In dynamic programming, we make use of temporarily stored intermediate results to bootstrap and extend for a new, more optimal iteration of an intermediate result until we have reached the desired solution.

Definition 6 *The **forward variable** $\alpha_t(i) = P[o_1...o_t|q_t = S_i, \lambda]$ is defined as the probability of observing the partial observation sequence $o_1...o_t$ given State S_i at timestep t as well as a fully parameterized model λ.*

We will use dynamic programming to compute the forward variable $\alpha_t(i)$ from our solutions for $\alpha_{t-1}(j)$ $1 \leq j \leq N$ (see Figure 6). The full solution

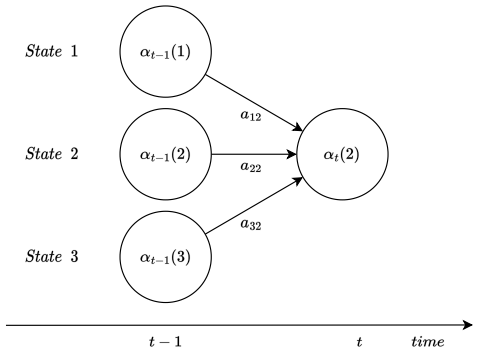


Figure 6: Usage of dynamic programming in the forward algorithm.

is as follows:

1. Initialization:

$$\alpha_1(i) = \pi_i \cdot b_i(o_1) \qquad 1 \leq i \leq N$$

2. Induction

$$\alpha_t(j) = \left(\sum_{j=1}^{N} \alpha_{t-1}(j) a_{ij} \right) \cdot b_j(o_{t-1}) \qquad 1 \leq j \leq N$$

$$1 < t \leq T$$

3. Termination

$$P[\mathcal{O}|\lambda] = \sum_{i=1}^{N} \alpha_T(i)$$

In the first step, we initialize the storage for the intermediate results, before continuously calculating the next intermediate results (see Figure 7). In the end, we simply sum over $\alpha_T(i)$ to obtain our desired result. This method of bootstrapping and reusing old results dramatically reduces the number of required operations down to N^2T (from the previous N^T) operations [16]. It should be noted, that computing $\alpha_t(i)$ for every t and i for a given observation sequence $\mathcal{O}$ yields the so-called posterior distribution for the hidden states given the observation sequence. This will be important later on when we will discuss the prediction capabilities.

2.5.2 Solution to Problem 2

One possible solution to the question "*What's the most likely state-sequence for observation $\mathcal{O}$?*" is to find the most probable state S_i for each timestep

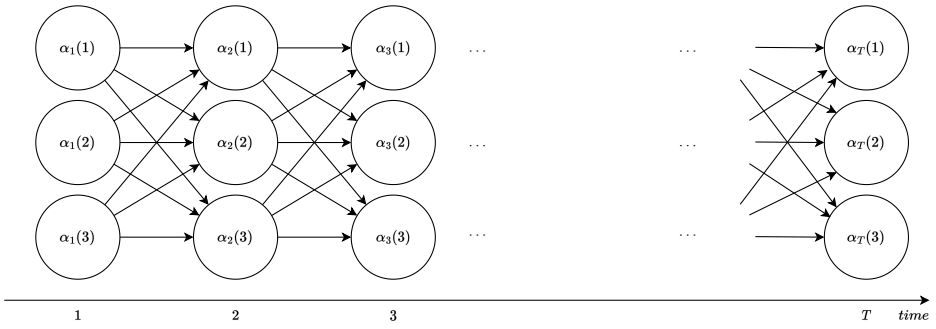


Figure 7: Visual representation of the full lattice structure used to compute $\alpha_T(i) \;\; 1 \leq i \leq N$.

t. To tackle this problem we will need further definitions, first of all, let us define the so-called "backward-variable", which in its definition is quite similar to the forward variable.

Definition 7 *The **backward variable** $\beta_t(i) = P[o_{t+1}...o_T|q_t = S_i, \lambda]$ is defined as the probability of observing the partial observation sequence $o_{t+1}...o_T$ given State S_i at timestep t as well as a fully parameterized model λ.*

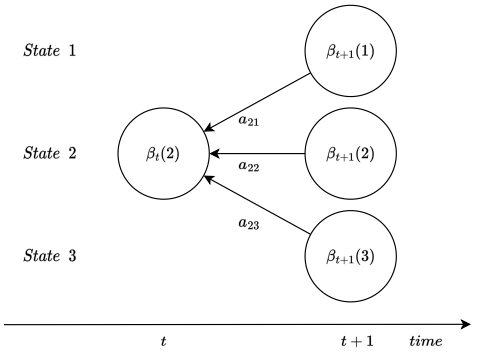


Figure 8: Usage of dynamic programming for the calculation of the backward variables.

To calculate the backward variable, we use the same idea of dynamic programming, reusing old results (see Figure 8). Although this time, we travel "backwards" through the observation sequence, thus starting at the last observation o_T.

1. Initialization:

$$\beta_T(i) = 1 \qquad 1 \leq i \leq N$$

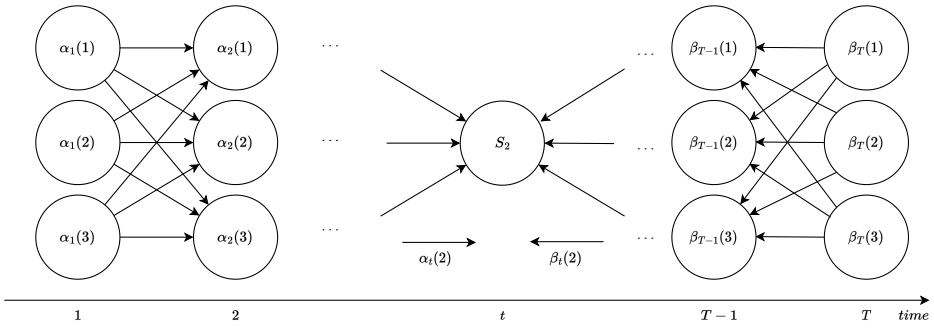


Figure 9: Usage of lattice structure in the forward-backward algorithm.

2. Induction

$$\beta_t(i) = \sum_{j=1}^{N} \beta_{t+1}(j) a_{ij} b_j(o_{t+1}) \qquad 1 \leq i \leq N$$

$$1 \leq t < T$$

Having both, the forward and the backward variable at our disposal, we can continue defining a helper variable $\gamma_t(i)$.

Definition 8 *The helper variable $\gamma_t(i) = P[q_t = S_i|\mathcal{O}, \lambda]$ denotes the probability of being in the hidden state S_i at timestep t given the full observation sequence $\mathcal{O}$ as well as a fully parameterized model λ.*

We can express the variable $\gamma_t(i)$ in terms of the forward and backward variables (see Figure 9).

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{P[\mathcal{O}|\lambda]} = \frac{\alpha_t(i)\beta_t(i)}{\sum_{i=1}^{N} \alpha_t(i)\beta_t(i)}$$

In this equation $P[\mathcal{O}|\lambda]$ is a normalization factor. Thus, $\sum_{i=1}^{N} \gamma_t(i) = 1$. To continue with our interpretation of optimality for a state sequence, we can solve for the individually most likely states for each timestep.

$$q_t = \underset{1 \leq i \leq N}{\arg\max}(\gamma_t(i)), \qquad 1 \leq t \leq T$$

Unfortunately, there is a flaw in this solution. Although we have found the individually most likely states, we have no guarantee for soundness of the found sequence of states. It might just be, that our sequence contains an illegal state transition. This error is resolved by the *Viterbi Algorithm* [18], which we will define in the following.

Definition 9 *The **Viterbi Algorithm** finds the most likely state sequence $Q = \{q_1, q_2, ..., q_T\}$ to have emitted a given observation sequence $\mathcal{O} = \{o_1, o_2, ..., o_t\}$, whilst respecting the state transition constraints given by the model λ.*

Definition 10 $\delta_t(i) = \max_{q_1,...,q_{t-1}} P[q_1, ..., q_t = i, o_1, ..., o_t|\lambda]$ *is the probability for the most probable state sequence along a single path $q_1, ..., q_t$ with observation sequence $o_1, ..., o_t$ up until timestep t, that ends in the hidden state S_i.*

Again, we can compute this variable inductively:

$$\delta_t(j) = (\max_{1 \leq i \leq N} \delta_{t-1} a_{ij}) \cdot b_j(o_t)$$

Since our goal is to find the optimal state sequence, we need a way to keep track of which prior state maximized $\delta_t(i)$ at each timestep t for each state i. This is done with a storage array $\psi_t(i)$. This lets us define the Viterbi Algorithm in four steps.

1. Initialization:

$$\begin{aligned} &\delta_1(i) = \pi_i b_i(o_1), && 1 \leq i \leq N \\ &\psi_1(i) = 0 \end{aligned}$$

2. Recursion:

$$\begin{aligned} \delta_t(j) &= \max_{1 \leq i \leq N} (\delta_{t-1}(i) a_{ij}) b_j(o_t) && 1 \leq j \leq N \\ & && 2 \leq t \leq T \\ \psi_t(j) &= \arg\max_{1 \leq i \leq N} (\delta_{t-1}(i) a_{ij}) && 1 \leq j \leq N \\ & && 2 \leq t \leq T \end{aligned}$$

3. Termination:

$$\begin{aligned} P^* &= \max_{1 \leq i \leq N} (\delta_T(i)) \\ q_T^* &= \arg\max_{1 \leq i \leq N} (\delta_T(i)) \end{aligned}$$

4. Backtracking: $q_t^* = \psi_{t+1}(q_{t+1}^*), \quad t = T-1, T-2, ..., 1$

Backtracking is a common method in computation, where we first compute our solution, storing information about the optimal path along the way before tracing back said path. A useful analogy might be laying breadcrumbs along a path through a forest of decisions. When we have found the right solution at the end of the forest, we want to know what path brought us here, hence tracking back our path by following the trail of breadcrumbs. In this context, we use this method to trace back the optimal states for our given observation sequence. This concludes the solution to problem 2.

2.5.3 Solution to Problem 3

Problem 3, which is concerned with "training" an HMM in a way such that the model is more likely to "explain" a given observation sequence, is the most difficult one out of the three presented problems. Starting from a rough estimate of the model parameters $\mathcal{A}, \mathcal{B}, \pi$, we can iteratively improve the model with the Baum-Welch Algorithm, which one might interpret as an application of the EM-Algorithm to HMMs [9]. Since the formal definition of the optimization would go out of scope for this thesis, the interested reader is referred to Rabiner's in-depth Tutorial on Hidden Markov Models [16].

2.6 Complexity Theory and Runtime-Analysis

In the context of Hidden Markov Models and the different algorithms used to compute various probabilities more or less efficiently, but also with the implementation section of this thesis in mind, I would like to give a short refresher on the most fundamental aspects of Complexity Theory.

In this thesis, the Bachmann-Landau Notation (or sometimes simply called big-O notation) is used to give upper bounds for the time complexity or runtime and memory intensity of algorithms with respect to one or many parameters. The big-O notation is a very convenient, compact way to describe the so-called "worst case" for runtime or memory consumption of a program, as well as allowing for easy comparison between different programs, regardless of the language they are formulated in.

We understand the time (or memory), that a program consumes before it terminates as a function of its input parameters. When we say "Program A solves task B in $\mathcal{O}(N)$-time.", we mean that given the input parameter N of task B, the time it takes for the program to finish the computation to solve the task is upwards-bounded by some (in this case) linear function of N. When N, the input parameter increases, the time required to solve the

task increases linearly. A formal definition [12] is given below.

Definition 11 *Let* $f : \mathbb{N}_0 \to \mathbb{R}_{\geq 0}$. *Then,* $\mathcal{O}(f) = \{g : \mathbb{N}_0 \to \mathbb{R}_{\geq 0} | \exists n_0 \in \mathbb{N}_0, c \in \mathbb{N} : g(n) \leq c \cdot f(n) \forall n \geq n_0\}$ *is the set of all functions* g *that are asymptotically upwards-bounded by* f. *We say* $g \in \mathcal{O}(f)$ *or that the growth rate of* g *is limited by* f.

Said plainly, we say that g is in $\mathcal{O}(f)$ (usually we use $\mathcal{O}(n)$ instead) if we can find a scalar $c \in \mathbb{R}_{\geq 0}$ with which scaled, the function $c \cdot f$ is asymptotically bigger than g.

Continuing towards runtime analysis, a brief example of the reasoning behind such analysis is supplied in the following paragraph. Given is a simple function written in the programming language Python [6].

```
import numpy as np

def some_other_function(N : int, M : int) -> float
    return (N + M) / 2

def construct_matrix(N : int, M : int) -> np.ndarray

    ret = np.zeros(shape=(N,M))

    for i in range(N):
        for j in range(M):
            ret[i,j] = some_other_function(N,M)

    return ret
```

One might ask the questions "*What is the time it takes for a function call* `construct_matrix` *to finish?*" or "*How much memory does the function call* `construct_matrix` *need?*". These kinds of questions can be answered using the big-O notation. In this particular case, we can easily see, that the function fills in a 2-dimensional array with the result of another function, namely `some_other_function`. Not only do we have to take into account the nested structure, but also the time and memory- complexity of the nested function call.

Let us start with `some_other_function`. We can see, that the function does some basic arithmetic operations, returning the computed value instantly. The time and memory complexity of this simple function is constant. In big-O notation, this is commonly denoted as $\mathcal{O}(1)$.

Having solved for time and memory complexity of the inner loops function call, we can continue analyzing the time and memory complexity of the original function of interest. As already stated, we are filling a 2-d array with values, each taking constant time to compute. Thus, our time and memory complexity are solely dependent on the parameters which define the shape of the said 2-d array. These parameters happen to be N and M our input parameters. Thus, we conclude that our program has the time and memory complexity $\mathcal{O}(N) \cdot \mathcal{O}(M) \cdot \mathcal{O}(1) = \mathcal{O}(N \cdot M)$. This analysis is important if we want to reason about what happens when we scale up the input parameters of our function.

2.7 Machine Learning Metrics

Metrics are an important tool in ML to quantify the success of a system for a given dataset. It is a measure of quality for the learned model. Additionally, metrics let us compare different models with different architectures. The metric used is dependent on the type of prediction that is to be made and since the following classification problem presented in this thesis is of a multi-class nature, we will motivate and explain one suitable metric for these classification problems - the multi-class F_1-Score.

Suppose we have a binary classification problem, where our task is to map an unknown data instance to one of two classes. Thus there are four thinkable outcomes for our prediction.

1. **True Positive (TP)**: We classify an instance of class 1 as class 1
2. **True Negative (TN)**: We classify an instance of class 2 as class 2
3. **False Positive (FP)**: We classify an instance of class 2 as class 1
4. **False Negative (FN)**: We classify an instance of class 1 as class 2

Imagine now increment a counter variable for each possible outcome of prediction, e.g. if we predict an instance of class 1 to be of class 1 correctly, we increment the variable v_{11}, if we instead incorrectly predict class 2, the variable v_{12} is incremented. This process is repeated for many unknown data instances. This kind of measurement can be represented in a so-called "confusion matrix" C, where the index c_{ij} corresponds to the variable v_{ij}.

Definition 12 *We call*

$$F_\beta = (1 + \beta^2) \cdot \frac{precision \cdot recall}{(\beta^2 \cdot precision) + recall}$$

the F_β-Score. For $\beta = 1$ we obtain the F_1-Score.

The F_1-Score is a common metric used to measure the quality of binary classifiers. It represents the harmonic mean between precision and recall, two values we can compute with the help of C, our confusion matrix.

$$precision = \frac{true\ \ positives}{true\ \ positives + false\ \ positives} = \frac{c_{11}}{c_{11} + c_{22}}$$

$$recall = \frac{true\ \ positives}{true\ \ positives + false\ \ negatives} = \frac{c_{11}}{c_{11} + c_{21}}$$

We can extend the F_1-Score to the multi-class domain, by computing a $K \times K$ confusion matrix, where K is the number of classes our model can predict. For each class K_i we can compute the counts for TP, TN, FP, FN as follows:

1. **TP**: C_{ii}
2. **FP**: $\sum_{j \neq i} C_{ji}$
3. **FN**: $\sum_{j \neq i} C_{ij}$
4. **TN**: $\sum_{j \neq i} \sum_{k \neq i} C_{kj}$

Thus, we can compute the F_1-Scores for each class. The final F_1-Score is either an average or a weighted sum of the respective F_1-Scores [11].

This yields a way for us to evaluate the performance of our model on a given dataset. Though, the problem of uncertainty remains. Since our dataset contains multiple samples, who decides which samples are used for training and which are used for the model evaluation (also called validation) process? We may begin by randomizing the dataset, then selecting the samples we would like to use for validation. Although this brings us one step further, it may still be that, by random, these chosen samples represent an unbalanced subset of the original dataset. To counteract this, we could repeat the process of shuffling data, selecting validation samples and evaluating the model on said validation samples many times. This is, what is generally referred to as "k-fold cross validation" [11]. In k-fold cross validation we split the given dataset k times into two separate datasets (usually in a 90-training to 10-validation ratio), one for training and the other for validation. The final overall resulting F_1-Score is then computed as the mean overall F_1-Scores evaluated for the individual train-validate dataset pairs. Finally, we are presented with a robust metric for evaluating our model for a given dataset. This concludes the section on Machine Learning Metrics.

3 Method

In the following section, I'll introduce the reader to the notation that will be used further on, as well as give an in depth model overview.

3.1 Notation

Definition 13 *A **marker** M is a unique identifier or class-name for a certain set of discrete states $S = \{S_1, ..., S_N\}$. We say that $\mathcal{M}$ is the set of all markers.*

Markers are the abstract handle for the observable units in our environment. For example, whilst observing a medical patient, the set of markers $\mathcal{M}$ could include "ability to walk", "blood pressure", "breathing frequency" and so on. The set of concrete states of the marker "blood pressure" could be $S = \{low, medium, high\}$. $\mathcal{M}$ is partitioned into a set of layers $\mathcal{L}$.

Definition 14 *A **(prediction-) layer** L is an element of $\mathcal{L}$, the partition of $\mathcal{M}$. A mapping $L_{M \to L} : \mathcal{M} \to \mathcal{L}$ maps a given marker M to its corresponding layer L.*

Using this definition we can partition the set of markers into layers containing somewhat related markers. For example, we could group the markers "ability to walk" and "number of pushups" into a layer "mobility". Next up, we should define the importance of these layers and their markers for our expected prediction. This is done by defining the following weight functions.

Definition 15 *Let a **layer-weight function** $\omega_{\mathcal{L}} : \mathcal{L} \to \mathbb{R}_{\geq 0}$ be a mapping from a layer L to a positive weight. The function $\omega_{\mathcal{L}}$ must satisfy $\sum_{L \in \mathcal{L}} \omega_{\mathcal{L}}(L) = 1$.*

Definition 16 *Let a **marker-weight function** $\omega_L : L \to \mathbb{R}_{\geq 0}$ be a mapping from a marker M of layer L to a positive weight. The function ω_M must satisfy $\sum_{M \in L} \omega_L(M) = 1$.*

Definition 17 *A **Trail** T is an observation sequence of consecutive states of length t, which belong to a marker M. The weight of a trail is defined as $\omega_T(T) = \omega_L(M) \cdot \omega_{\mathcal{L}}(L_{M \to L}(M))$.*

Definition 18 *An **Observation** $\mathcal{O} = \{T_1, ..., T_k\}$ is a set of Trails of same length t. The mapping $M_{T \to M} : \mathcal{O} \to \mathcal{M}$ maps a Trail T to its corresponding marker M.*

Definition 19 *A **Query** $\mathcal{Q} = (M_{\mathcal{H}}, \mathcal{L}, \mathcal{O}, \{\omega_{L_1}, ..., \omega_{L_k}\}, \omega_{\mathcal{L}})$ consists of a hidden marker $M_{\mathcal{H}}$ as well as a partition of $\mathcal{M}$, namely $\mathcal{L}$. A Query possesses an observation $\mathcal{O}$, consisting of possibly many trails T. The weights of the markers and layers are defined by the marker-weight functions ω_{M_i} (one for each layer) and the layer-weight function ω_L.*

We have established the fundamental building blocks for our prediction model. As the reader might have extracted from the notation given above, a Query $\mathcal{Q}$ is a well-defined description of a multivariate sequence alongside mixture components, namely the weights of the markers and layers. The mixture components are the weights, by which the individual result of each HMM is weighted. We would like to continue to define a formalism, which maps augmented versions of these queries to a prediction result. The problem is, that the form of the prediction is dependent on the augmented query. Plainly said, our model can predict time series of discrete states, as well as time series of distributions over states. At first, we will keep the prediction result - in any case, some sort of time series - very abstract.

Definition 20 *A **trail-evaluation***

$$\phi(T, M_{\mathcal{H}}) = \omega_T(T) \cdot P(T, \lambda(M(T), M_{\mathcal{H}}))$$

is a function that maps a Trail, a Query and possibly various arguments to a specifically requested weighted prediction result $P(T, \lambda(M(T), M_{\mathcal{H}}))$.

Here, $\lambda(M(T), M_{\mathcal{H}})$ is a function, that returns a fully parameterized HMM whose hidden states are the states of the marker $M_{\mathcal{H}}$ and the observable states are the states of the trail marker $M(T)$. Subsequently, the resulting HMM is used to reason about the given trail T. Further details about the nature of the prediction are given in Section 3.3.4.

Definition 21 *An **observation-evaluation***

$$\Phi(\mathcal{Q}) = \sum_{T \in \mathcal{Q}} \phi(T, M_{\mathcal{H}}), \qquad M_{\mathcal{H}} \in \mathcal{Q}$$

is a function that sums up the weighted prediction results to construct the final prediction.

As we can see, the result of the evaluation of an observation under a Query is the weighted sum of the evaluations of the trails - the constituents of the observation.

3.2 Model Overview

Observing an environment in the real world allows for the observation of the states of multiple markers M_i at each timestep. Certain semantically related markers are grouped into layers. We would like to reason about a hidden state in this environment based upon our observations by constructing a Query $\mathcal{Q}$. The idea is to construct an HMM for every single marker and try to infer knowledge about the hidden states from given observations, according to the weight of the marker $\omega_L(M)$ and the weight of

its respective layer $\omega_{\mathcal{L}}(L(M))$. The resulting model can briefly be described as a mixture of Hidden Markov Models - one HMM for each marker. It is important to understand, that our model is only able to offer prediction capabilities that a simple one-observation HMM could offer as well.

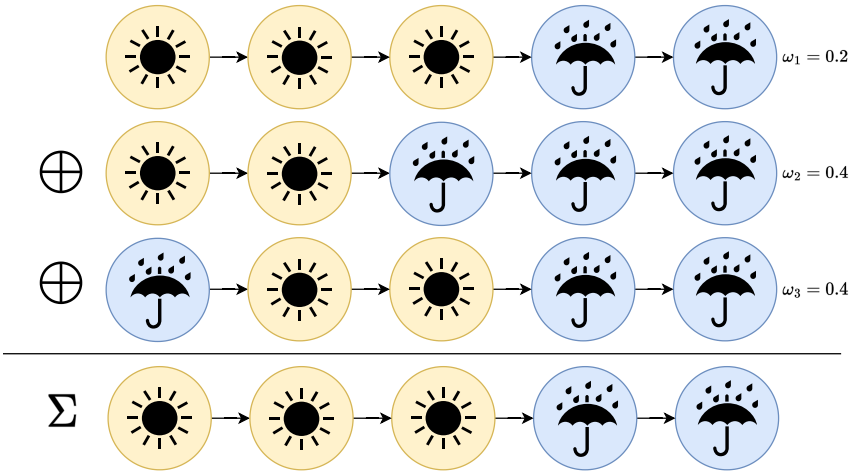


Figure 10: Conceptual depiction of the workings of the prediction model. The model can roughly be described as a mixture of HMMs.

In this thesis, we assume the hidden marker to be one of the visible observation markers, allowing for the ability to generically switch the hidden marker and prediction-layer to allow for maximum flexibility. Hence, we will be able to extract the parameters of the model, namely $\mathcal{A}, \mathcal{B}$, and π from the observation sequences directly, instead of having to train the model based on said sequences. This is in stark contrast to the usual usage of HMMs, where these parameters are not given and have to be approximated (learned) from observations.

3.3 Pipeline Description

The prediction pipeline was written in Python, making use of the `hmmlearn` library [3]. The pipeline is made up of a pre-processing stage, a feature extraction stage, and a prediction stage (see Figure 11). To obtain a trained model, the user must input their training data in the form of a .csv file as well as a .ini config file, describing the layer structure and provide sufficient information about the weights of the model.

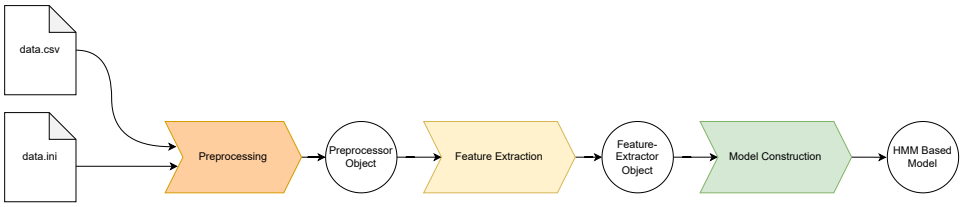


Figure 11: Flow-diagram of the prediction pipeline. The data is converted to a fully functioning model.

3.3.1 User Input

The data consisting of the different observations made in the form of a .csv file, as well as the configuration of the model in the form of a .ini file, must be supplied by the user. The .ini file must conform to a given abstract syntax (see Appendix A) which we will discuss in the following.

The abstract syntax outlines the correct way of specifying a .ini file required to construct a model, whilst making no assumption about the form of the data. It is written in a metalanguage called "Backus Naur Form" or BNF for short [13]. Each marker in the data which the user wants to be integrated into the model must be specified as a section inside the .ini file. Additionally, information about the datatype, related layer, and layer-specific weight of the marker must be supplied as a key-value pair under the corresponding markers section. Optional information, like the relationship to other markers, can be added.

An important aspect of the data is the grouping of observations and the measurement interval. The user can specify the group key or primary key of the data, by which each observation instance is identified, as well as a maximum time delta in between two consecutive measurements under the meta-information section as key-value pairs. Omitting both key-value pairs will result in an interpretation of the data as one large observation sequence, where the consistency of the length of the time intervals in between observations is unimportant. Of course, this is not advised, as the time interval between observations does matter a lot for most cases. Inconsistencies in this regard are sure to distort results, or at least lessen the value of any made prediction by the model.

To give a short example, a .csv file and its corresponding .ini file have been provided in the appendix section of this thesis (see Appendix B). In the example, we have two observation sequences, one observed by henry, and the other one observed by scarlet. Both have marked their observations for different markers in the corresponding columns and added a date of observation. In the corresponding .ini file (see Appendix B) we have specified by

which column to group the observations under the `markerconfig_metainfo` section. Additionally, we have connected the markers **Snow** and **Temp** to the **weather**-layer and the markers **Chocolate Type** and **Xmas Lights** to the **goodies**-layer. For each marker, we have specified a datatype, notably there exists a linspace datatype which given the fitting parameters specified in the abstract syntax is able to discretize continuous or already discrete values into categories. This feature is especially important since this kind of discretization allows us to generalize and enable the HMM to work with continuous values it has never seen before. The abstract syntax also allows for discretization of timestamps, which is just another example of a continuous datatype.

It should be noted that, although the definition of a marker calls for the existence of a layer-specific weight, the program is robust against missing or faulty weights and will re-balance the given weights to satisfy stochastic constraints.

3.3.2 Pre-processing

Pre-processing is a modular stage inside the pipeline, which itself is a small pipeline (see Figure 12). The transformation of the data supplied includes the analysis of the .ini file, deletion of any unnecessary data, the grouping of the data according to the metainformation extracted from the .ini file, enforcing measurement interval consistency if necessary, and finally encoding the data into a less memory intensive format.

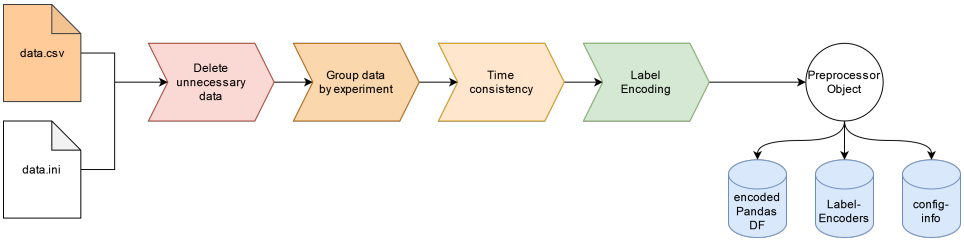


Figure 12: Flow-diagram of the pre-processing pipeline.

The label encoding transformation is a standard procedure ensuring a stable workflow as well as providing a point of standardized contact with the following components (or interface for short), at which the single previous or latter components of the pipeline might be easily switched out or modified. This practice ensures modularity, besides reducing the amount of memory used to store the observation sequences. For convenience, the whole pre-processing is fully automated and can be called in a few lines of code:

```
from pre-processing import Preprocessor

path_to_config = "./data.ini"
path_to_data = "./data.csv"

prep = Preprocessor(debug=False)
prep.process(path_to_config=path_to_config,
             path_to_data=path_to_data,
             csv_delimiter=',')
```

3.3.3 Feature Extraction

The Feature Extraction builds upon the previous step in the pipeline, the pre-processing (see Figure 13). Just as the prior component of the pipeline, the feature extraction component is fully modularized implementing the necessary interface used to provide the required functionality to the next part in the pipeline. Inside the feature extraction stage, the **state transition**-, **signal emission**- and **initial state**-probabilities are extracted from the encoded data supplied by the pre-processing stage. This is an important step since we can use the extracted probabilities later on to construct HMMs with a strong initial guess for $\mathcal{A}, \mathcal{B}$ and π.

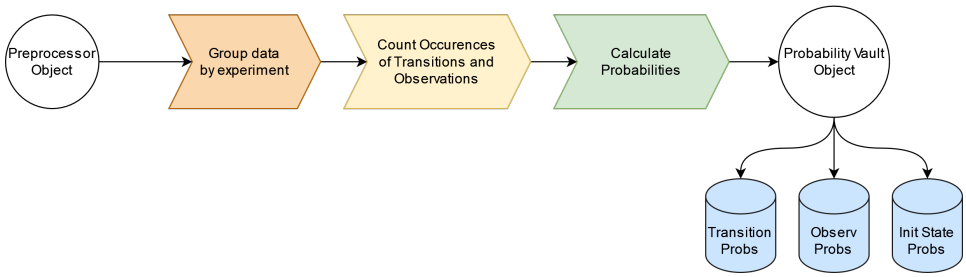


Figure 13: A flow-diagram of the Feature Extraction Pipeline.

At this point, I would like to take the opportunity to talk about the memory complexity of the feature extraction. Naturally, we would like to extract the transition probability matrices for every marker M from the data. In the usual case the state transition matrix $\mathcal{A}$ should be sparse, meaning the diversity in transitions between states is very limited. In other words, we do not expect every state to have a transition probability above 0 to many other states, especially with a high number of states. Additionally, we have to extract an emission matrix $\mathcal{B}$ for every state for every other marker. Let us make a rough estimation of the memory complexity of such an extraction. Suppose we have N markers, with S states each. The memory complexity for the state transition matrices alone is $\mathcal{O}(N \cdot S^2)$ since we would

have to reserve a $S \times S$ transition matrix for each marker. To make matters worse, let us take a look at the emission matrices. Here, we would have to reserve $\mathcal{O}(N \cdot (N-1) \cdot S^2)$ memory since for every state of every marker we would have to account for every other emission signal (state) of every other marker. We can immediately see, that a static allocation of memory is infeasible for our needs.

Alternatively, we could think about computing $\mathcal{A}, \mathcal{B}$ and π at query time. This, although certainly running shivers down many computer scientists' spines, is the least memory intensive method for making a prediction. Here, the problem isn't the memory intensity, it's the time complexity of the extraction. Let us assume the query $\mathcal{Q}$ with hidden marker $M_{\mathcal{H}}$ with $S_{\mathcal{H}}$ hidden states, as well as the prediction-layers $L_i,\ 1 \leq i \leq M$. Furthermore, assume that each layer L_i has N markers, each with S states. To allow prediction, we must now compute the state transition matrix $\mathcal{A}$ and π for $M_{\mathcal{H}}$ as well as the emission matrices for the $M \cdot N$ observation markers. Although looping over all observation data at query time could technically be done in $\mathcal{O}(T)$ time, where T is the number of observations, this is merely the tip of the iceberg. We are left having to construct the stochastic matrices which includes counting occurrences of states and normalization as well as having to perform the standard error checks. In summary, the time complexity comes out to be (roughly) $\mathcal{O}(T \cdot M \cdot N + M \cdot N \cdot S \cdot S_{\mathcal{H}})$, which one should not compute at query time for $T \gg 1$ or large numbers of N, M or S.

The problem becomes balancing the time and memory complexity in order to avoid both pitfalls. The solution chosen in this thesis builds upon the assumption, that state transitions are sparse, as well as the general paradigm to push most computational work into the pre-processing section of the program, saving much computational cost at query-time. To count state occurrences, dictionaries (or HashMaps) are used, in order to minimize the space needed to store information. The worst case memory complexity remains, but it should be noted that relying on this kind of dynamic allocation of memory is far more memory efficient and reasonable than statically allocating tons of memory at the beginning of pre-processing. Normalization and the usual checks for the satisfaction of stochastic constraints are done in-place inside the dictionaries. This yields all necessary state transition-, emission- and initial state-probabilities. The actual matrices are constructed at query time. This poses no direct problem, as the bulk of the work is already done. The computational effort required to read from a dictionary and write into $N \cdot M$ matrices is negligible. This concludes the feature extraction section of the pipeline. Again, for convenience, the feature extraction can be written in a few lines of code.

```
from pvault import ProbabilityVault

# build ontop of the already existing Preprocessor object
pv = ProbabilityVault(prep, debug=False)
pv.extract_probabilities()
```

3.3.4 Model Validation and Query

Finally, in the last step of the pipeline, the HMM-based model is constructed and queried. Building on top of the previously constructed feature extraction, the actual construction of the different HMMs is very convenient. In the implementation, we heavily rely on the hmm-learn library [3] and its implementation of the Multinomial Hidden Markov Model. All that is left is the interpolation of the results given by the individual HMMs according to the weights specified inside the .ini config-file. To give a measure of success, the model is able to compute the multi-class F_1-Score for a given validation dataset. Model construction and validation can be executed in a few lines of code, as we will see below.

```
import pandas as pd
from model import RHMM

path_to_validation_data = "./validation.csv"

# load validation dataset, split into groups
validation_df = pd.read_csv(path_to_validation_data, delimiter=','
                                    )
validation_samples = prep.group_df(validation_df)


# build model on top of already contructed ProbabilityVault object
rhmm = RHMM(pv, debug=False)
f1_score = rhmm.validate(   groups=validation_samples,
                            layers=['layer1', 'layer2'],
                            hidden_marker='marker1')
```

As already stated, it is important to understand that the HMM-based model can only offer predictions that a simple single observation HMM could offer as well. These predictions include the following:

1. **Posteriors for each hidden state for observation $\mathcal{O}$**
   Given a Query $\mathcal{Q}$, compute the posterior distribution for each hidden state of $M_{\mathcal{H}}$ for every timestep t given the trails T_i. The result will be a weighted sum (according to the weights defined in $\mathcal{Q}$) of the individually computed posteriors.

2. **Distribution over hidden states following $\mathcal{O}$**
   Given a Query $\mathcal{Q}$, predict the distribution over the hidden states of $M_{\mathcal{H}}$ for possibly many timesteps $\hat{t}$ following the observation. This yields an approximation to a stationary distribution of the state transition matrix for $M_{\mathcal{H}}$. The kind of stationary distribution is dependent on the initial state distribution given by the observation sequence $\mathcal{O}$.

3. **Optimal state sequence**
   Given a Query $\mathcal{Q}$, compute the optimal state sequence of $M_{\mathcal{H}}$ "best explaining" the trails T_i using the Viterbi Algorithm.

3.3.5 Controller

To wrap all of these components up, and use the whole pipeline, as well as enable plotting of the results, a wrapper object called `Controller` provides a user friendly interface.

```
from controller import Controller

# construct model
c = Controller()
c.construct(path_to_data="data.csv",
            path_to_config="data.ini",
            csv_delimiter=",")

# validate model
c.validate(path_to_validation_data="validation.csv",
           csv_delimiter=",",
           hidden_marker="marker1",
           layers=["layer1", "layer42"])

# query model
c.plot_posterior_distribution(path_to_observation="single.csv",
                              csv_delimiter=",",
                              hidden_marker="marker1",
                              layers=["layer1"])
```

In most cases, we would like to test the performance of our model on a specific dataset more rigorously. This is archived with the so-called **k-fold cross validation**, a way of reducing uncertainty over the performance of a model on a dataset. The model is continuously trained and tested on randomized parts of the whole dataset, resulting in multiple measurements of model performance. The Controller provides a simple method interface for the k-fold cross validation of a given dataset.

```
from controller import Controller

path_to_data = "data/train.csv"
path_to_config = "data/train.ini"

c = Controller()

f1_scores = c.kfold_cross_validation(k = 10,
                            path_to_data=path_to_data,
                            path_to_config=path_to_config,
                            csv_delimiter=',',
                            layers=["layer1", "layer2", "layer3"],
                            hidden_marker='hidden_marker')
```

For the interested reader, a fully functional version of the proposed pipeline will be provided in the form of a tutorial jupyter-notebook as part of the open source repository [1] to provide an easy stepping stone for inexperienced users and enable reproduction of the following results.

3.4 Tools

In the following paragraph, I'll briefly examine the tools used for this thesis, highlighting the advantages of each tool, the thought process behind choosing the respective tool and improvements that could be made in hindsight.

A convenient way of keeping track of implementation issues and tasks was the software repository management system GitHub [2]. GitHub allows for great planning, project management and version control, far beyond the needs of this thesis. In my case, GitHub was mainly used to keep track of issues and have a remote copy of the working repository. As the repository was initialized, the next step was to prototype the individual parts of the pipeline. For this, jupyter-notebooks [4], a combination of an interactive python shell and a code editor, was used as it allows for quick feedback on whether an idea is working or not. Once done prototyping, an actual python IDE (Integrated Development Environment) in the form of PyCharm [5] was used to ease development and find bugs. At this stage of the project, all code was moved from loosely hanging notebooks into an object oriented structure. In hindsight, the process of transferring notebook-code to python-scripts could have been done earlier. Here, more time should have been invested in actually planning what sort of application was to be developed and the requirements for the fluid development of such an application.

4 Model Evaluation

The model was evaluated on two different datasets, firstly on a for this thesis synthesized dataset, secondly on the DZNE dataset containing real medical data. Since the dataset of the DZNE contains sensitive personal data, the extent to which the results can presented here is quite limited, which is why I'll provide an in-depth look at the prediction and evaluation for the first, synthesized dataset.

4.1 Data Generation

As already stated, the first dataset is synthesized and its sole purpose is to give the reader a good understanding of the capabilities of the presented prediction model. The dataset mimics the actual DZNE dataset in which we expect to see many markers with degenerative states - markers, whose states continuously degenerate over time going from an initial good state into a worse state. Four different degenerative markers were selected, namely

1. $M_{motoric}$, a marker whose state indicates the subjects ability to solve tasks using their hands
2. $M_{mobility}$, a marker whose state indicates the subjects state of mobility, e.g. still being able to walk properly
3. M_{neuro}, a marker whose state indicates the subjects ability to solve mental tasks
4. $M_{diagnosis}$, a marker whose state indicates the diagnosis given by an expert for a subject at a certain timestep

Every marker has the same set of degenerative states, namely $S = \{$good, med-good, med, med-bad, bad, severe $\}$ indicating the current state under the given marker. To construct the data, a state transition matrix $\mathcal{A}$ as well as an initial state distribution vector π were constructed for the single marker $M_{diagnosis}$. Additionally, emission signal matrices ($\mathcal{B}_{motoric}$, $\mathcal{B}_{mobility}$, $\mathcal{B}_{neuro}$) were constructed for the other markers $M_{motoric}$, $M_{mobility}$, M_{neuro}. Using these parameters, three different HMMs were constructed. Using the initial state probability distribution, for each observation sequence, an initial state for the marker $M_{diagnosis}$ was chosen. Thereafter, the following states for $M_{diagnosis}$ were sampled from the state transition probability matrix $\mathcal{A}$. Likewise, for each timestep of each observation, the emission signal for each observation marker M_i was sampled from the distribution located in the respective emission signal matrix $\mathcal{B}_i$. A dataset of $N = 300$

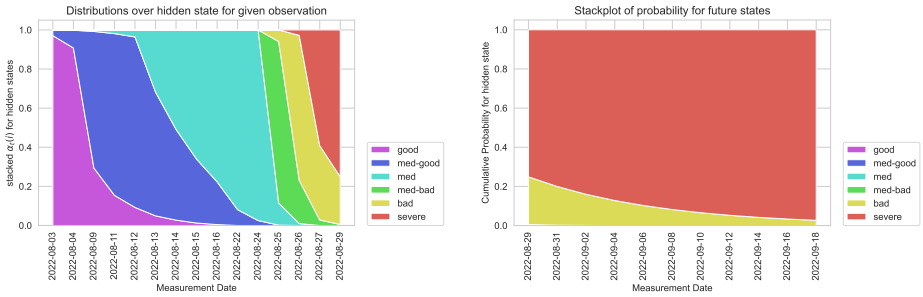


Figure 14: Posteriors for the given hidden states (left) and the extrapolated distribution (right) under $M_{\mathcal{H}} = M_{diagnosis}$.

observation sequences of variable sequence-length and variable time intervals in between measurements was synthesized. To finalize the data, a configuration .ini file had to be constructed. For simplicity, each marker was assigned to a distinct layer, resulting in four layers - one for each marker.

4.2 Model Evaluation of the Generated Dataset

The dataset was evaluated with a 10-fold cross validation. The layers of Query $\mathcal{Q}$ were set to $L = \{$ L_1 = mobility, L_2 = motoric, L_3 = neuro $\}$. In this special case, every marker has an equal influence on the model prediction, since every layer consists of only one marker and the layer weights are equally distributed across all layers. Alternatively, the layer weights $w(L_i)$ could be adjusted or multiple markers could be grouped in one layer, allowing for further differentiation by adjusting the marker weights $w(M_i)$ inside one layer. The hidden marker $M_{\mathcal{H}}$ was set to the marker $M_{diagnosis}$.

In plain English, the model query could be summarized as trying to predict a state sequence of diagnoses given the observation of the patients mobility-, motoric- and neuro-markers.

As already stated above, the model was evaluated with a 10-fold cross validation which yielded an average ***F_1*-Score of 0.813 with a standard deviation of 0.017**. To showcase the prediction capabilities of the model further, a never seen before observation was generated and the model was queried for all three possible prediction types (see Section 3.3.4). These include the prediction of the posteriors and the extrapolation given an observation sequence (see Figure 14) as well as the prediction of an optimal state sequence (see Appendix C).

4.3 Background and Description of the DZNE Dataset

The German Center for neuro-degenerative Diseases, or DZNE for short provided a dataset obtained by an observational study of subjects suffering from a hereditary neuro-degenerative disease called "Spinocerebellar ataxia type 3" (SCA3). Symptoms include progressive loss of balance, coordination deficits and slurred speech. SCA3 patients experience significant restrictions of mobility and communicative skills. Preventive interventions that aim to silence the disease gene offer a promising treatment option and the first clinical gene silencing trial[1] has recently started. Consequently, there is an urgent need to predict the deterioration to a more severe disease stage to prioritize and treat patients suffering from a greater risk with less uncertainty.

The supplied data contains results from various tests, the subjects had to take repeatedly over an extended period of time. The tests themselves assign a "health-rating" (from 0 to 4 in most cases) to special kinds of abilities or measured observations of the subjects, where a rating of 0 indicates a healthy observation. Higher ratings are applied, when the subject starts to perform worse in a given section of a test. Notably, the "ADL" (Activities of Daily Life), "INAS" (Intentional Non-Adherence Scale) and "SARA" (Scale for the Assessment and Rating of Ataxia) scales were considered in the first part of the experiment. These specific scales were chosen after consulting with a domain expert. With the additional information about the - by domain experts already inferred - disease state for each multivariate observation in our data, we return to the familiar setup presented in the prior section, where we have multiple Trails paired with a given diagnosis or inferred latent state. The difference is, that instead of the 3 observation markers in the prior example, we get about 109 different markers, which we can observe at every timestep. The average length of the given time-series was calculated to be about 2.3 timesteps.

4.4 Model Evaluation on the DZNE Dataset

4.4.1 First Experiment

The 109 markers were partitioned into three layers, L_{ADL}, L_{INAS} and L_{SARA}. The weight function ω_L assigns each prediction-layer the weight $\frac{1}{3}$. Internally, the weight function responsible for assigning a weight to every marker inside a given layer assigns equal weight to every marker. The hidden marker $M_{\mathcal{H}}$ was chosen to be the diagnosis marker. This proved to be a naive approach, as the bad performance of the model with a sub .5 F_1-Score in a 10-fold cross-validation of the whole dataset reassured.

[1] ClinicalTrials.gov, Identifier: NCT05160558

F_1-Scores for 10-fold cross validation ($\mu \pm \sigma$)				
layers \ $M_{\mathcal{H}}$	diagnosis	ADL score	INAS score	SARA score
$L_{ADL}, L_{INAS}, L_{SARA}$	0.75 ± 0.04	0.22 ± 0.06	0.13 ± 0.04	0.19 ± 0.06
L_{SARA}	0.75 ± 0.03	0.23 ± 0.03	0.16 ± 0.04	-
L_{INAS}	0.6 ± 0.06	0.11 ± 0.04	-	0.13 ± 0.04
L_{ADL}	0.72 ± 0.08	-	0.19 ± 0.05	0.24 ± 0.04

Table 1: Table cells display the mean and standard deviation of the F_1-Scores received from a 10-fold cross validation. Different constellations of prediction-layers and hidden marker were used.

4.4.2 Second Experiment

Since first experiments yielded an unacceptable performance, the decision was made to reduce the number of Trails, by only considering the cumulative score for each of the three scales (ADL, INAS and SARA), drastically reducing the number of Trails down to only three. It was here, that the model was able to perform relatively well, with an **F_1-Score with a mean of 0.75 and a standard deviation of 0.04** as the result of a 10-fold cross validation on the whole dataset. Additionally, further constellations of prediction-layers and hidden markers were tested (see Table 1). The predictions with $M_{\mathcal{H}} =$"diagnosis" generally produced acceptable results, whereas the other predictions yielded rather unsatisfying prediction results.

4.5 Interpretation and Discussion

Generally, it was expected that the model would perform relatively well given the prediction right prediction-layers and enough training data. A substantial difficulty with the DZNE data was, that a majority (approx. $\frac{2}{3}$) of the given training sequences was of length 1 or 2, thus the dataset lacked sufficient samples for longer time-series. Of course, this also resulted in the validation dataset mostly containing short sequences. I would like to point out, that although we have instances of multivariate time-series in the DZNE dataset, the actual shape of the data makes it rather unsuitable for an HMM-based prediction, simply because the real strength of HMMs, which is sequence to sequence prediction, cannot be fully harnessed. However, it was shown in Section 4.4.2, that we can reach reasonable prediction scores using an HMM approach - at least for the right constellations of prediction-layers and hidden marker.

The reasonable results produced, given the prediction-layers L_{ADL}, L_{INAS}, L_{SARA} in constellation with the hidden marker $M_{\mathcal{H}}$ set to the marker "diagnosis" can be explained by the strong correlation between the individual markers inside the prediction-layers (which are simply the scores for the corresponding scale) and the diagnosis given by a medical professional. Specifically, the prediction-layer L_{SARA} seems to be best suitable for producing predictions for the hidden marker "diagnosis". This independently received finding correlates with the findings of domain experts, i.e. the medical experts of the DZNE.

The question left to ask is why we couldn't do better. There are multiple explanations for this, which I would like to discuss. Firstly, the most dominant reason is the length of the time-series we base our prediction on, as well as the distribution for the initial hidden state. As already described, about $\frac{2}{3}$ of the training data contains time-series only of length 1 or 2. This greatly influences the prediction capabilities for longer time-series. The dataset simply doesn't have enough training samples for longer time-series, thus the performance for predicting longer sequences is worse. Additionally, since we extract the probabilities from the training data, the model has a strong bias for the probability of the initial state. Without knowing anything about the validation sample, a certain initial state is far more probable than another. This fact, paired with the appearance of short (sometimes single timestep) time-series is most definitely a source for errors in our prediction. Furthermore, a thorough analysis of the individual time-series and the corresponding scales has shown, that the underlying Markov Chains for the markers are not strictly left-right. To put it in other words, the difference in e.g. the ADL score, a measure of the subjects ability to manage activities of daily life, for the last and first measurement is not strictly positive. In some cases, we can observe, that a patient reduces their score over time, instead of displaying a degenerating performance (see Appendix D).

In the case of predicting the hidden markers "ADL Score", "INAS Score" or "SARA Score" the granularity of the discretization of the continuous scales deeply impacts the prediction score. For the given data, the granularity (i.e. the dtype linspace defined inside the .ini file) was kept the same for all measurements. In practice, the user would of course be able to change the number of categories, the continuous values for each scale can fall into, to reduce the number of misclassifications. Simple tests showed, that the terrible performance can be improved to an F_1-score of about 0.5 by reducing the number of categories from about 20 down to 5. Surprisingly, doing this has a negative effect on the prediction results for the marker "diagnosis", as it seems an over-simplification does harm to the precision and recall of

our prediction.

5 General Discussion

In the following section, I'll discuss the presented Model, its strengths as well as its weaknesses. Furthermore, I'll discuss how one could improve or augment the prediction capabilities of the model.

5.1 Strengths, Weaknesses and Improvements

First, let us discuss the abilities of the model. Since the model can be seen as a mixture of HMMs, its capabilities to capture deep complexity and intricate relationships within data is very limited. As the model can use multiple markers for its prediction, the general stability of a prediction is traded for a lack of attention to details in the data. Since we weigh the influence the markers have on the prediction separately, small but maybe very critical changes in a single marker might be overpowered by the sheer number of markers our model has to respect. The only way to combat this problem right now would be to assign a higher weight to critical markers, in order to let small changes in states for an important marker have a greater impact on the final prediction. This however requires domain expertise and is generally not in the spirit of Machine Learning. It would be much more convenient and interesting to let an algorithm find such weights itself. This could be done by employing an optimization technique such as gradient based optimization or perhaps a probabilistic technique such as a genetic algorithm. The optimization problem then becomes arbitrarily difficult, as we could think about not only optimizing the weights for the layers and markers but optimizing weights for each timestep of the given time-series independently. It might just be that the importance of the state of a marker changes over time, which we would then be able to observe with the change of the weight of the marker over time. This of course would scale up the effort required for an optimization drastically, depending on the length of our time-series.

A crucial topic we should not miss to talk about, is the way in which we have worked with HMMs in the context of the data provided. Generally, Hidden Markov Models do model, as their name suggests, hidden states, their initial state probabilities, transitions and emission signal probabilities. The usual case is that these three parameters are learned or approximated, as already stated in Section 3.2. However, we have assumed an observable state to be our hidden State. This was necessary in order to maintain the

flexibility and generality of the model, although of course this is not the usual use-case of HMMs. Subsequently, we were able to omit the step of training an HMM on our data, since we were already provided with precise measurements inside our training data and didn't have to approximate which parameters might have led to the displayed data - we could extract the data right away. Alternatively, we could think about using the extracted probabilities as a starting point for our optimization. As the training of HMMs is very sensitive to the choice of the initial guess for the parameters [16], another optimization idea would be to 1) extract the probabilities for $\mathcal{A}, \mathcal{B}, \pi$, 2) possibly apply some sort of smoothing to the parameters and 3) then let an optimization algorithm find a local optimum for said parameters. This was presented by [8], who optimized their HMM with a genetic algorithm.

5.2 Model Capabilities and Usecases

Another topic I want to touch on is the gain, which the presented prediction pipeline brings for domain experts like medical professionals from the DZNE. During the design of the pipeline, it was kept in mind that people lacking a computer science background should be able to use the tool with minimal additional effort required. The prediction target and result can not only be altered by specifying the prediction-layers and hidden marker, but also by specifying the weight functions for the layers and their respective markers as well as the partition of the markers into layers in the first place. Domain experts are free to adjust these parameters within the constraints of the proposed abstract syntax (A). Apart from configuring the models parameters, domain experts are presented with multiple prediction capabilities, whose applications we would like to discuss separately:

5.2.1 Posteriors for States of a Hidden marker

Our prediction tool is able to predict and plot posteriors for the states of a hidden marker $M_{\mathcal{H}}$. For a given observation sequence, the model predicts $\alpha_t(i)$ for every state i and every timestep t of the observation sequence. This enables us to produce a simple visualization of how likely the model thinks a certain hidden state is at a given timestep of the observation. A domain expert can see at a glance in which direction the states of $M_{\mathcal{H}}$ evolve.

5.2.2 Approximation of the Stationary Distribution

Additionally, we can give a rough estimate of the "future" posteriors of the states of $M_{\mathcal{H}}$ beyond the given observation sequence. In other words, we can give an estimate about how the distribution over the probability for

the states of $M_{\mathcal{H}}$ will evolve over future timesteps. It should be added, that this estimate is a visualization of the approximation of the stationary distribution of the state transition matrix $\mathcal{A}$ of the underlying Hidden Markov Model.

5.2.3 Optimal State Sequence Prediction

The most valuable prediction capability might be the prediction of optimal state sequences given an observation sequence. This allows for "double-checking" already found observation sequences and might be used for data augmentation in the case of missing data or that a malfunctioning sensor gives back erroneous measurements. Predicting an optimal sequence of states for a hidden marker "diagnosis" might perhaps be a help to a medical professional, who wants to verify their given diagnosis.

5.2.4 Extraction of Model Parameters

Finally, we can extract the model parameters themselves as they can give us critical information about the general transition probabilities of a model. A domain expert might ask about a rough estimate of the probability for state transitions, which we could immediately answer by extracting the state transition probability matrix $\mathcal{A}$ from our model.

6 Summary

In this thesis, I presented both theoretical and practical foundations for a generically generated HMM-based prediction model. In the course of the thesis, the model was tested on medical data provided by the DZNE[2] and achieved a F_1-score of 0.75 in a 10-fold cross validation.

I introduced new notation in Section 3, which serves to precisely describe the theoretical foundations such as the type of data processed or the weighting of a prediction. Special emphasis was put on the definition of a query $\mathcal{Q}$, which can be mapped to a prediction result by the theoretical model represented by a function P.

Following this, I described the individual modular components of the prediction pipeline (pre-processing, feature extraction as well as model validation and query) throughout Section 3.3 and presented an interface that enables the user to generate an HMM-based model from their data.

An interactive jupyter notebook [1] was supplied for any interested reader who might want to explore further possibilities or attempt to repeat the presented experiments. Subsequently, I applied my approach first on a synthetic dataset and second on a real medical dataset provided by the DZNE in Section 4, where the presented model was able to achieve a F_1-Score of 0.75 when predicting diagnosis based on clinical measures.

The independently obtained finding that the prediction-layer L_{SARA}[3] can be best used for a prediction of the diagnosis is in agreement with experiences of domain experts of the DZNE and acts as proof of concept for the approach presented in this thesis. In the future, the quality of a prediction could be further increased primarily with the increasing availability of balanced training data and secondarily with an optimization of the parameters of the model - either by domain experts or through algorithmic optimization techniques.

[2]The DZNE is the German center for neurodegenerative diseases, based in Bonn, Germany

[3]SARA (Score for the Assessment and Rating of Ataxia) is a medical measure, which is to be interpreted as an indicator for a patients medical state.

A Abstract Syntax for the Configuration File

```
// This is the abstract syntax defining the .ini configuration file syntax
// Please note that the general structure of the .ini syntax is not changed,
// we merely need to define some required key/value pairs,
// as well as some sections.

<Config> ::= [ <Meta> ] <Sections>

<Meta> ::= "[markerconfig_metainfo]" [ <layerInfo>] [ <Groupby>] [ <Dateinfo> ]

// The layerinfo is a disctionary containing a mapping
// from layername to corresponding layerweight.
<layerInfo> ::= "layerinfo" "=" <layerWeightMap>

// The Groupby key is the columns name of a unique identifier
// of an observation sequence like an uID or a unique name of a subject
<Groupby> ::= "groupby" "=" <SectionIdentifier>

// Dateinfo contains the column name of the date column, as well as
// two additional states. The delta between the additional dates
// will be interpreted as the maximum timedelta inbetween two measurements
<Dateinfo> ::= "dateinfo" "=" "(" <SectionIdentifier> "," <Date> "," <Date> ")"

// Please specify the layerWeightMap as you would a python dictionary
// (String keys, Float values)
<layerWeightMap> ::= "{" <layerWeights> "}"
<layerWeights> ::= <layerWeight> | <layerWeight> "," <layerWeights>
<layerWeight> ::= <layerIdentifier> ":" <Float>
<layerIdentifier> ::= <String>

<Sections> ::= <Section> | <Section> <Sections>
<Section> ::= "[" <SectionIdentifier> "]" <Attributes>
<SectionIdentifier> ::= <String>

// Attributes can be required Attributes or optional Attributes.
<Attributes> ::= <ReqAttributes> [ <OptAttributes> ]
<ReqAttributes> ::= <Datatype> <layerName> <layerSpecWeight>
<OptAttributes> ::= <OptAttribute> | <OptAttribute> <OptAttributes>
<OptAttribute> ::= <RelationList>

<Datatype> ::= "dtype" "=" ("discrete" | <Continuous>)
<layerName> ::= "layerName" "=" <layerIdentifier>
<layerSpecWeight> ::= "layerSpecificWeight" "=" <Float>

<RelationList> ::= "relations" "=" "[" <Relations> "]"
<Relations> ::= <Relation> | <Relation> "," <Relations>
<Relation> ::= "(" <RelationType> "," <SectionIdentifier> ")"
```

```
<RelationType> ::= <String>

// Please see
// https://numpy.org/doc/stable/reference/generated/numpy.linspace.html
// for more information about how to define this datatype.
// This dtype can be used to discretize continuous values into bins, in
// order to transform the data at hand into categorically distributed data.
<Continuous> ::= "linspace(" <Float> "," <Float> "," <Integer> ")"

// Date should be defined, according to ISO8601,
// Float, Integer, String are defined as usual.
<Date> ::= <Placeholder>
<Float> ::= <Placeholder>
<Integer> ::= <Placeholder>
<String> ::= <Placeholder>
<Placeholder> ::= "Please infer right side from type name"
```

B Example .csv and .ini file

Person	Date	Snow	Temp	Chocolate Type	Xmas Lights
henry	2020-12-01	yes	-0.2	none	1
henry	2020-12-02	no	2.1	nougat	3
henry	2020-12-03	no	3.0	white	5
henry	2020-12-04	yes	1.0	nougat	4
scarlet	2020-12-06	yes	-0.3	nougat	3
scarlet	2020-12-08	yes	0.2	none	2
scarlet	2020-12-02	yes	-1.2	white	1
scarlet	2020-12-12	yes	-3.0	cacao	14
scarlet	2020-12-24	yes	-1.0	dark	42

Table 2: Table view of an exemplary csv file containing multivariate time-series. Two series of measurements are made. One is recorded under the uID "henry", the other under the uID "scarlet".

Contents of an exemplary .ini configuration file are displayed. The syntax conforms to the abstract syntax definition defined in Appendix A.

```
[markerconfig_metainfo]
groupby=Person
dateinfo=('Date', '2020-12-24', '2020-12-25')

[Person]
dtype=discrete
layerName=personal
layerSpecificWeight = 1

[Date]
dtype=discrete
layerName=measurement
layerSpecificWeight = 1

[Snow]
dtype=discrete
layerName=weather
layerSpecificWeight = 0.5

[Temp]
dtype=linspace(-10,10,20)
layerName=weather
layerSpecificWeight = 0.5

[Chocolate Type]
dtype=discrete
layerName=goodies
layerSpecificWeight=0.8

[Xmas Lights]
dtype=linspace(0,50,10)
layerName=goodies
layerSpecificWeight=0.2
```

C Optimal Path Prediction

Optimal State Sequence Prediction		
Timestep	$M_{\mathcal{H}}$ (ground truth)	$M_{\mathcal{H}}$ (predicted)
1	good	good
2	good	good
3	med-good	med-good
4	med-good	med-good
5	med-good	med-good
6	med-good	med
7	med	med
8	med	med
9	med	med
10	med	med
11	med	med
12	med-bad	med-bad
13	med-bad	bad
14	severe	severe
15	severe	severe

Table 3: Prediction result of an optimal state prediction of the HMM-based model. The values for the ground truth are displayed on the left, whilst the prediction results are displayed on the right side. Faulty classifications are marked in red.

D Distributions for Scales

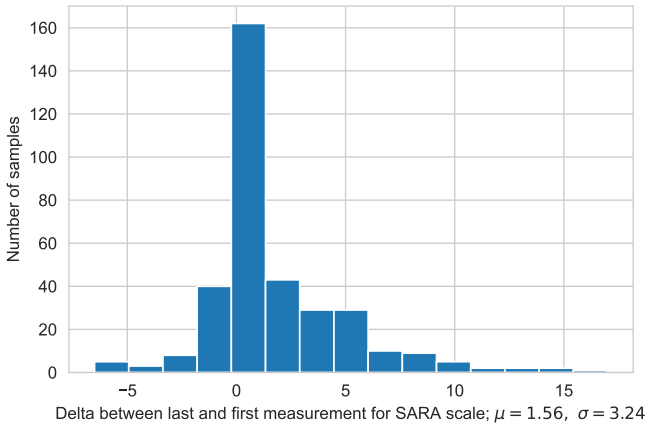


Figure 15: Distribution for the difference in measurements for the SARA (Scale for the Assessment and Rating of Ataxia) scale. For each sample, the delta between the last and the first measurement was taken. A positive delta is a degeneration, a negative delta shows improvement according to the given scale.

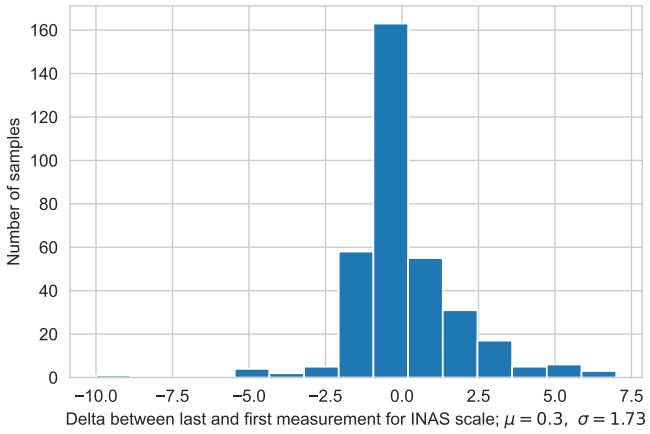


Figure 16: Distribution for the difference in measurements for the INAS (Intentianal Non-Adherence Scale). For each sample, the delta between the last and the first measurement was taken. A positive delta is a degeneration, a negative delta shows improvement according to the given scale.

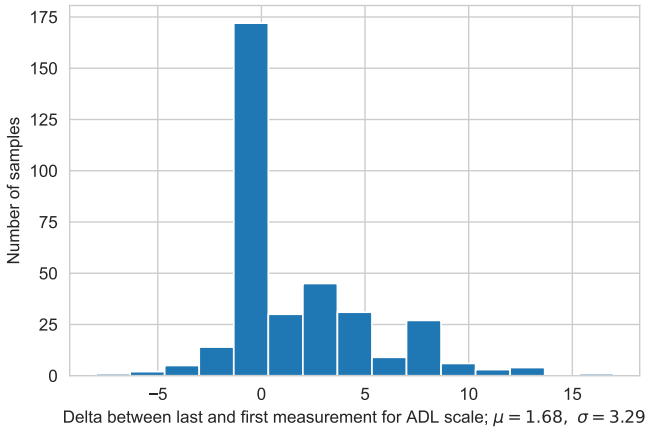


Figure 17: Distribution for the difference in measurements for the ADL (Activitied of Daily Life) scale. For each sample, the delta between the last and the first measurement was taken. A positive delta is a degeneration, a negative delta shows improvement according to the given scale.

References

[1] generic hmm pipeline. https://github.com/rfechner/generic-hmm. Accessed: 2022-07-29.

[2] github. https://github.com/. Accessed: 2022-07-29.

[3] hmm-learn library. https://hmmlearn.readthedocs.io/en/latest/. Accessed: 2022-07-27.

[4] jupyter. https://jupyter.org/. Accessed: 2022-07-29.

[5] pycharm. https://www.jetbrains.com/de-de/pycharm/. Accessed: 2022-07-29.

[6] Python programming language. https://www.python.org. Accessed: 2022-07-27.

[7] James Baker. The dragon system–an overview. *IEEE Transactions on Acoustics, speech, and signal Processing*, 23(1):24–29, 1975.

[8] Chak-Wai Chau, Sam Kwong, CK Diu, and Wolfgang R Fahrner. Optimization of hmm by a genetic algorithm. In *1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 1727–1730. IEEE, 1997.

[9] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22, 1977.

[10] Paolo Frasconi, Giovanni Soda, and Alessandro Vullo. Text categorization for multi-page documents: A hybrid naive bayes hmm approach. In *Proceedings of the 1st ACM/IEEE-CS joint conference on Digital libraries*, pages 11–20, 2001.

[11] Margherita Grandini, Enrico Bagli, and Giorgio Visani. Metrics for multi-class classification: an overview. *arXiv preprint arXiv:2008.05756*, 2020.

[12] Rodney R Howell. On asymptotic notation with multiple variables. *Tech. Rep.*, 2008.

[13] Donald E Knuth. Backus normal form vs. backus naur form. *Communications of the ACM*, 7(12):735–736, 1964.

[14] Hyeon-Kyu Lee and Jin-Hyung Kim. An hmm-based threshold model approach for gesture recognition. *IEEE Transactions on pattern analysis and machine intelligence*, 21(10):961–973, 1999.

[15] Mikael Nilsson and Marcus Ejnarsson. Speech recognition using hidden markov model, 2002.

[16] Lawrence R Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[17] Srinivasan Vairavan, Larry Eshelman, Syed Haider, Abigail Flower, and Adam Seiver. Prediction of mortality in an intensive care unit using logistic regression and a hidden markov model. In *2012 Computing in Cardiology*, pages 393–396. IEEE, 2012.

[18] Andrew Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE transactions on Information Theory*, 13(2):260–269, 1967.