Petar Veličković

Molecular multiplex network inference

Project Dissertation
Computer Science Tripos, Part II

Trinity College

16 October 2015

Proforma

Name:	**Petar Veličković**
College:	**Trinity College**
Project Title:	**Molecular multiplex network inference**
Examination:	**Computer Science Tripos, Part II, 2015**
Word Count:	**11974**[1]
Project Originators:	**Dr Pietro Liò** and **Petar Veličković**
Supervisor:	**Dr Pietro Liò**

Original Aims of the Project

The primary aim of this project is to investigate the potentials of utilising complex networks to handle multiple types of correlated data simultaneously when making inferences in the context of binary classification problems. In order to do this, a generic machine learning library that uses multiplex networks was to be implemented and evaluated; the evaluation consists of assessing the performance benefits of this approach on synthetic and biomolecular data sets (utilising standard machine learning evaluation metrics), in particular compared to the single-layer versions, operating on a single type of data only.

Work Completed

The project has been highly successful; all of the main success criteria have been met and many extensions (additional models to test on, methods of assessing robustness in the presence of experimental errors, representing the network in a format suitable for visualisation tools, etc.) have also been implemented. The proposed machine learning library has been fully implemented from scratch using C++ and evaluated in the manner outlined above, using an evaluation suite which I have also implemented. The produced implementation of multiplex networks outperforms the single-layered version on both representative models implemented, with respect to standard classifier evaluation metrics.

Special Difficulties

None.

[1] The word count has been computed by TEXcount, http://app.uio.no/ifi/texcount/.

Declaration

I, Petar Veličković of Trinity College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed:

Date: 16 October 2015

Contents

Acknowledgements

The work presented in this dissertation, as well as the writeup itself, has been greatly facilitated by the contributions and suggestions of the following people, to whom I owe particular thanks:

- **Dr Pietro Liò**, for considerable efforts in supervising this project and guiding me towards a successful finished product;
- **Hui Xiao**, for providing significant assistance in interpreting and analysing the biomolecular data used for evaluation of my project;
- **Dr Arthur Norman** and **Dr Sean Holden**, for all of their support, comments and guidance throughout my studies;
- **Nikola Jovanović**, for investing considerable time in helping me proofread this dissertation, and providing a vast amount of constructive comments on its overall clarity and content.

Chapter 1

Introduction

The main aim of my CST Part II project has been to investigate and evaluate the potentials of using multilayer networks (specifically, multiplex networks) in machine learning problems, for handling multiple types of correlated data simultaneously when making inferences. In order to achieve this, a machine learning library for multiplex networks has been fully implemented and evaluated on several models and types of data.

In this chapter, I will discuss the primary motivations for implementing a data structure of this kind, outline the main challenges I was faced with during the implementation, and present a survey of related work in the area.

1.1 Motivation

With the development of experimental methods and technology, we are able to reliably gain access to data in larger quantities, dimensions and types. This has great potential for the improvement of machine learning (as the learning algorithms have access to a larger space of information). However, conventional machine learning approaches used thus far on single-dimensional data inputs are unlikely to be expressive enough to accurately model the problem in higher dimensions; in fact, it should generally be most suitable to represent our underlying models as some form of **complex networks**—graphs with nontrivial topological features.

Within the scope of this dissertation, a special kind of complex networks known as *multiplex networks* will be considered; informally, a multiplex network is a multi-layered graph in which each layer is built over the same set of nodes, and there may exist edges between nodes in different layers (a more formal definition will be given in §2.1). This model's suitability arises from the fact that there exists a wide variety of systems exhibiting "natural" multiplexity: social interactions (either physically or via social networks) [10, 24, 26, 34], transportation networks [8, 12], biochemical and genetic networks [40] and research communities [9], to name a few.

In order to showcase why such a method of representing data might be favourable, I will

briefly present a common motivating example of *epidemics and awareness* previously used in several academic papers. Another common motivating example, the analysis of *social network interactions* through multiplex networks, has been recently investigated by researchers from the Computer Laboratory [21].

1.1.1 Example: Epidemics and awareness

This example is due to Granell *et al.* [20], and attempts to model the behaviour of modern society in the presence of an *epidemic*. The authors consider two layers of interaction between people in such a scenario:

- The *epidemics* layer, where two people are connected if they are in physical contact (family, close friends, co-workers, etc.). A *susceptible* (S) person may become *infected* (I) with probability β if it is in contact with an infected person, and an infected person may spontaneously recover with probability μ.
- The *information* layer, where two people are connected if they regularly share information among each other (e.g. social network acquaintances). It has similar dynamics to the epidemics layer, with each node being represented as either *aware* (A) or *unaware* (U) of the epidemic, and changing state with probabilities $\lambda \sim \beta$ and $\delta \sim \mu$.

In addition, the layers are allowed to influence one another:

- A person that is infected may become aware of the disease with probability κ (*self-awareness parameter*);
- A person that is aware of the disease may take precautions (vaccination, protective masks, etc.) which effectively makes their probability of infection $\beta^A = \gamma\beta$ for a given parameter $\gamma \in [0, 1]$ (*immunisation parameter*).

Finally, the phenomenon of *mass media* in promoting awareness of the epidemic is modelled by a parameter m, implying that an unaware person may become aware with probability m, not by interacting with another person, but from an entity that globally transmits information (e.g. TV/radio stations, online blogs, etc.).

This kind of model lends itself naturally to a multiplex network representation; the two layers of interaction correspond to two graph layers, and each node is "connected" to itself in the other layer through the parameters κ and γ. Mass media can be additionally modelled as a "master node" connected to each node in the information layer through the parameter m. The full multiplex network is presented in Figure 1.1[1].

[1]This figure has been originally used in the Granell *et al.* [20] paper, and is re-used here with kind permission of the author.

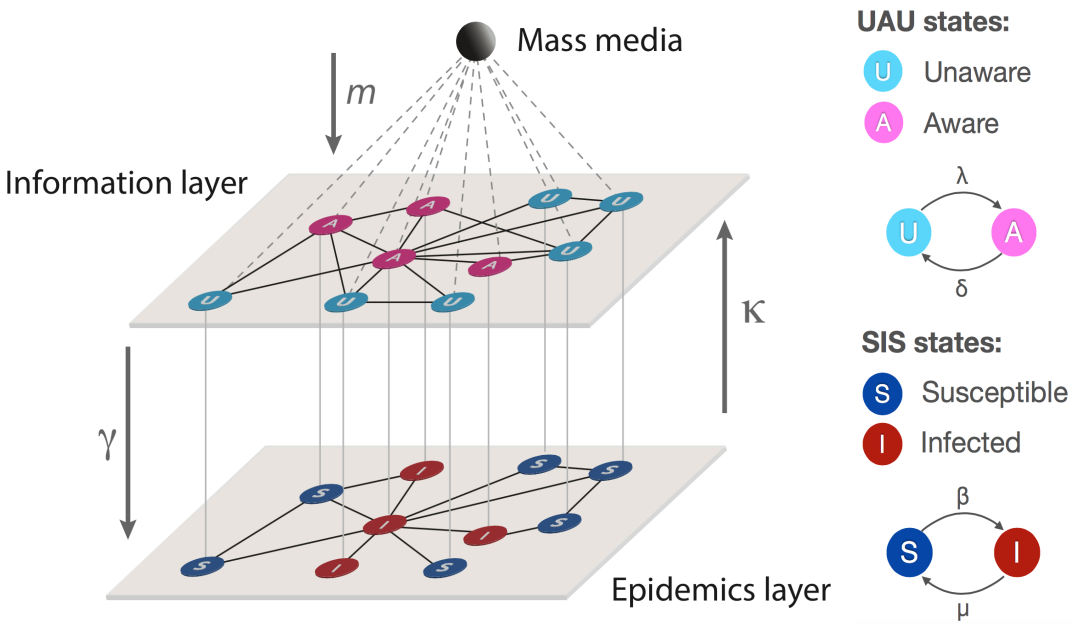


Figure 1.1: Example application of multiplex networks: Awareness-epidemic model in the presence of mass media.

1.2 Challenges

At the onset of this project, I had no prior knowledge of probabilistic machine learning models or multiplex networks; as such, a major initial challenge involved reading up on large amounts of relevant background material, so I could be properly set to tackle the project. In addition, research in complex networks is a relatively new trend, which meant that there was a lack of textbooks that could be utilised; I had to obtain most of the knowledge in this area by invesetigating relevant academic papers.

The main implementation goal of the project has been to adapt known machine learning algorithms and data structures to take advantage of multiplexity; because I wanted to make this as generic as possible and because integrating my project with existing library implementations might be cumbersome and not even work as proposed, I made a decision early on to make the project completely self-contained, implementing all the relevant single-layer machine learning algorithms myself. This resulted in a large codebase, and several subtle bugs and errors had to be taken care of during the implementation stage. Most of them have stemmed from the fact that the implemented algorithms are generally numerical in nature, and a minor error when working with floating-point types can gain much larger significance on the larger scale model's outputs.

1.3 Related work

1.3.1 Current research

Research in multiplex networks is currently highly active, and large quantities of papers are continuously published over the past few years investigating their various features [1, 6, 32], applications to modelling real-world systems [17, 20, 22, 44] or developing novel algorithms on them [2, 25].

1.3.2 Multilayer networks library

This library[2] builds on the theoretical foundations of multilayer networks outlined in Kivelä *et al.* [23], and implements a basic model of multilayer networks and computing various known metrics on them.

As the library is implemented in Python, it is unsuitable for larger data sets, and the algorithms it currently implements are not useful for the kind of problem this project aims to solve. At the time of writing this dissertation, it is, however, the only publicly available generic library implementation of multiplex networks I was able to find.

1.3.3 `muxViz`

`muxViz`[3] [11] is a software for multiplex network visualisation, utilising R, GNU Octave and OpenGL to produce high-quality 3D representations of the networks.

As such visualisations could be very useful for investigating the networks' properties, one of the extensions of the project I have implemented "dumps" the generated multiplex network into a format `muxViz` can directly read and analyse.

[2] http://www.plexmath.eu/?page_id=327
[3] http://muxviz.net

Chapter 2

Preparation

In this chapter, I will summarise all the preparatory work done before the implementation phase began. To begin, a summary of the initial study phase, in the form of an overview of the theory behind the two prinicipal data structures used in this project (multiplex networks and hidden Markov models), will be presented. The remainder of the chapter will focus on the project planning phases, imperative for successfully executing a project of this magnitude: formally defining the problem to solve, requirements and dependency analysis, choice of tools and software engineering practices.

2.1 Introduction to multiplex networks

Within the scope of the project proposal (Appendix D), I have given a 'self-contained' definition of multiplex networks and suggested a possible method of representation for them. In order to be fully precise about this, in this section I will formally define multiplex networks, by first defining their more generalised counterpart, *multilayer networks*, and general constraints that could be imposed on them—once that is done, defining multiplex networks reduces to specifying the required constraints.

2.1.1 Multilayer networks

Before properly defining multilayer networks, it is sensible to start off by revisiting the definition of **graphs** [7], as the main building blocks used to construct them.

Definition 1. A *graph* (within this context sometimes called a *single-layer* or *monoplex* network) is an ordered pair $G \stackrel{\text{def}}{=} (V, E)$, where V is a set of *nodes* and $E \subseteq V \times V$ is a set of *edges* that connect pairs of nodes together.

To extend this notion to multilayer networks, we will retain the concept of a set of nodes V over which the network is built; a **layer** $G_\alpha \stackrel{\text{def}}{=} (V_\alpha, E_\alpha)$ of this network can then be defined as a graph over a *subset* of nodes $V_\alpha \subseteq V$; by Definition 1 it must hold that $E_\alpha \subseteq V_\alpha \times V_\alpha$. A multilayer network consists of a **sequence of** L **layers**, $\mathcal{L} \stackrel{\text{def}}{=} \{G_\alpha\}_{\alpha=1}^{L}$.

A node in a multilayer network is uniquely determined by the member of the set

V it corresponds to, and the layer it's located in. It is hence reasonable to consider **node-layer pairs** when defining multilayer networks; the set of all such pairs, $V_M \subseteq V \times \mathcal{L}$ is trivially defined as $V_M \stackrel{\text{def}}{=} \{(x, G_\alpha) \mid x \in V_\alpha\}$.

Lastly, the node-layer pairs may be arbitrarily pairwise connected through **edges**; we define the **set of edges** between node-layer pairs $E_M \subseteq V_M \times V_M$, analogously as before. Using all of the previously defined elements, it is possible to define a multilayer network as follows:

Definition 2. A *multilayer network* is a 4-tuple $M \stackrel{\text{def}}{=} (V_M, E_M, V, \mathcal{L})$, where V is a set of nodes contained in the network, $\mathcal{L}$ is the sequence of layers the network consists of, and V_M and E_M are the sets of node-layer pairs and edges between them, respectively.

Essentially, the overall multilayered structure still represents a graph, which can be obtained by extracting the first two elements from the tuple; $G_M \stackrel{\text{def}}{=} (V_M, E_M)$.

The edges of a multilayer network are usually partitioned into two groups, depending on whether they're contained within a single layer:

- The set of *intra-layer* edges, E_A, linking together nodes within a single layer; $E_A \stackrel{\text{def}}{=} \{((x, G_\alpha), (y, G_\beta)) \in E_M \mid \alpha = \beta\}$. Note that this set is completely defined by $\mathcal{L}$; that is, $((x, G_\alpha), (y, G_\alpha)) \in E_M \iff (x, y) \in E_\alpha$.
- The set of *inter-layer* edges, E_C, linking together nodes from different layers; $E_C \stackrel{\text{def}}{=} \{((x, G_\alpha), (y, G_\beta)) \in E_M \mid \alpha \neq \beta\}$.
  - It is also helpful to consider a subset of inter-layer edges called the *coupling* edges, $E_{\widetilde{C}} \subseteq E_C$, linking nodes to their own images in different layers; $E_{\widetilde{C}} \stackrel{\text{def}}{=} \{((x, G_\alpha), (y, G_\beta)) \in E_C \mid x = y\}$.

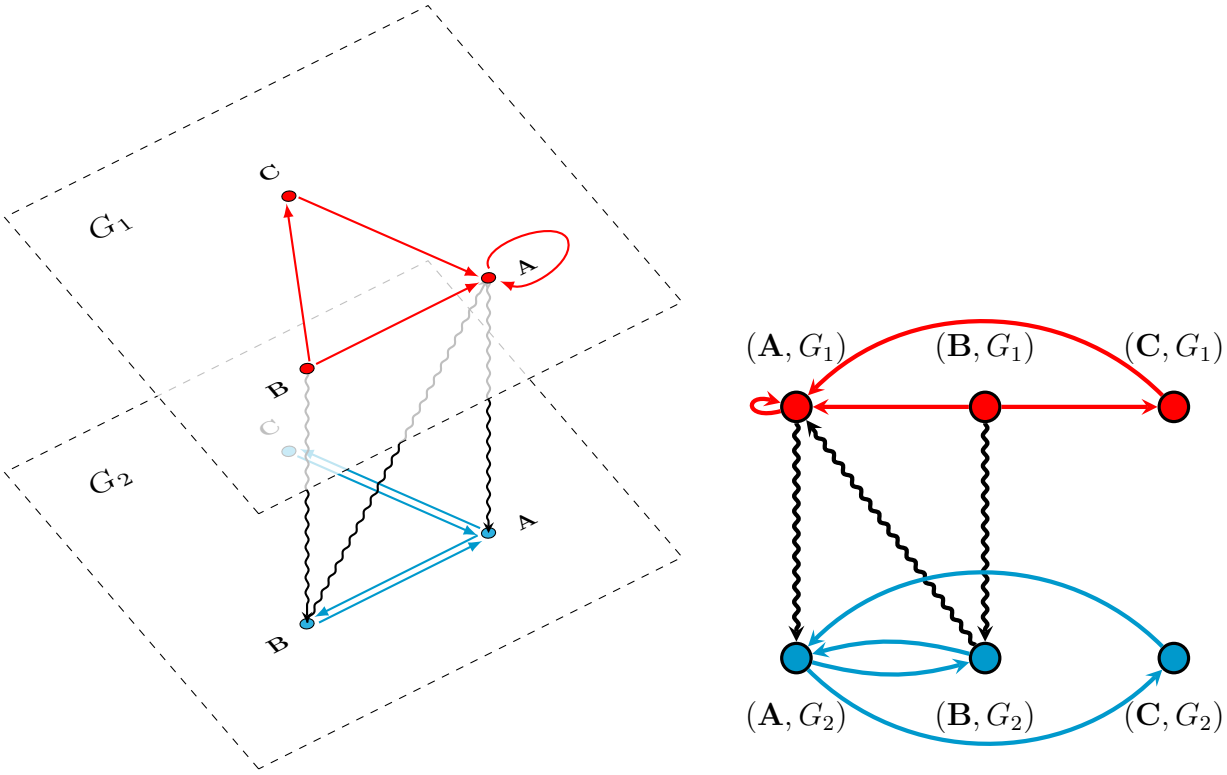
An example representation of a multilayer network (as well as its underlying graph, G_M) is given in Figure 2.1.

After defining all of the previous elements, it is possible to define constraints that may be imposed on a multilayer network's structure.

2.1.2 Constraints

Some of the constraints that may be imposed on a multilayer network are as follows [23]:

- A multilayer network is *node-aligned* if every layer contains all the nodes; i.e. $\forall G_\alpha = (V_\alpha, E_\alpha) \in \mathcal{L}.\ V_\alpha = V$. The network given in Figure 2.1 is node-aligned; both layers (G_1 and G_2) contain all three nodes ($\{\mathbf{A}, \mathbf{B}, \mathbf{C}\}$).
- A multilayer network is *diagonally coupled* if all inter-layer edges are coupling, i.e. $E_C = E_{\widetilde{C}}$. The network given in Figure 2.1 is not diagonally coupled, because the edge $((\mathbf{B}, G_2), (\mathbf{A}, G_1))$ is not coupling.



(a) Multilayer network, $M = (V_M, E_M, V, \mathcal{L})$

(b) Underlying graph, $G_M = (V_M, E_M)$

Figure 2.1: Example of a multilayer network (a) and its underlying graph (b).

- A diagonally coupled multilayer network is *categorically coupled* if each node is connected to *all* of its images in the other layers; i.e. $\forall x \in V.\ \forall G_\alpha, G_\beta \in \mathcal{L}.\ (x, G_\alpha) \in V_M \wedge (x, G_\beta) \in V_M \wedge \alpha \neq \beta \implies ((x, G_\alpha), (x, G_\beta)) \in E_M$.

2.1.3 Multiplex networks

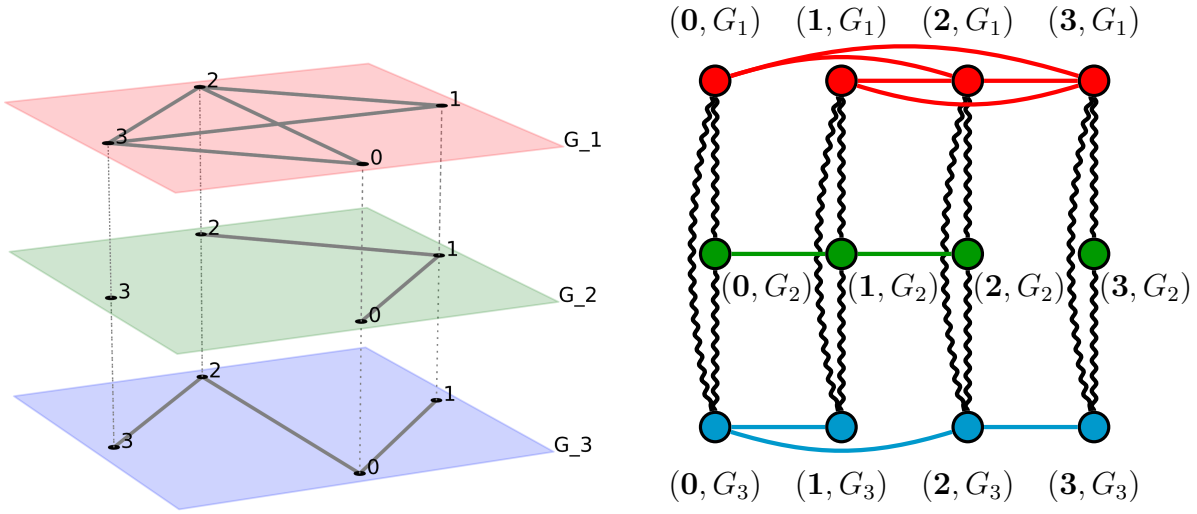
With the previously defined framework, it is simple to formally define a multiplex network, which also concludes this section.

Definition 3. A *multiplex network* is a node-aligned, diagonally coupled multilayer network.

From Definition 3 it follows that, to define an instance of a multiplex network, it is sufficient to specify V, $\mathcal{L}$ and $E_{\widetilde{C}}$. Furthermore, if the coupling is of a special type, $E_{\widetilde{C}}$ is no longer needed (this is the case for e.g. the previously mentioned *categorical* coupling). Figure 2.2 represents an example categorical multiplex network.

2.2 Gaussian mixture hidden Markov models

Having assimilated the material on the various kinds of multilayer networks, I was prepared to attempt to apply the multiplexity phenomenon to a machine learning model. After researching several alternatives, I opted for **hidden Markov models** (*HMM*s) [4, 5], for two primary reasons:



(a) Categorical multiplex network, $M = (V, \mathcal{L})$ (b) Underlying graph, $G_M = (V_M, E_M)$

Figure 2.2: Example of a categorical undirected multiplex network[2] (a) and its underlying graph (b).

- They are a fairly simple model to comprehend and implement, but have been proven to be powerful for many pattern-recognition tasks [35, 39, 41];
- Combining multiple HMMs with the use of a multiplex network lends itself to a relatively simple implementation and a straightforward justification (as will be described in §3.3.2.1).

Hidden Markov models are a principal component of several CST Part II courses' syllabi (*Artificial Intelligence II*, *Bioinformatics*, *Natural Language Processing*...), and as such their formal definition details are omitted from the main dissertation body; a thorough discussion is given in Appendix A.1.

This project utilises an extension to HMMs—namely, the *Gaussian mixture* HMM (*GMHMM*). The need for such a model has arisen primarily from my intent to handle *continuous outputs*; I opted for this approach because most of the experimentally observable data (including the biomolecular data designated for evaluation) is continuous—an implementation like this could prove to be more easily *reusable* (without having to modify much, if any, code) for scientists in a variety of different fields.

A GMHMM consists of a set of states, S, a set of "sub-outputs", O'. The system follows a particular sequence of states (with a *transition probability matrix*, $\mathbf{T}$, specifying the probabilities of transitioning between each pair of states), with each state emitting a sub-output, which in turn emits an output value that may be observed. Therefore, each state has associated with it the probabilities of emitting each sub-output (defined by an *emission probability matrix*, $\mathbf{O}'$), and each sub-output y' has its own parameters

[2]The multiplex network has been generated and drawn using the *multilayer networks library* (§1.3.2).

for a Gaussian distribution (mean $\mu_{y'}$ and standard deviation $\sigma_{y'}$) used to produce its corresponding outputs. Figure 2.3 depicts an example of a GMHMM.

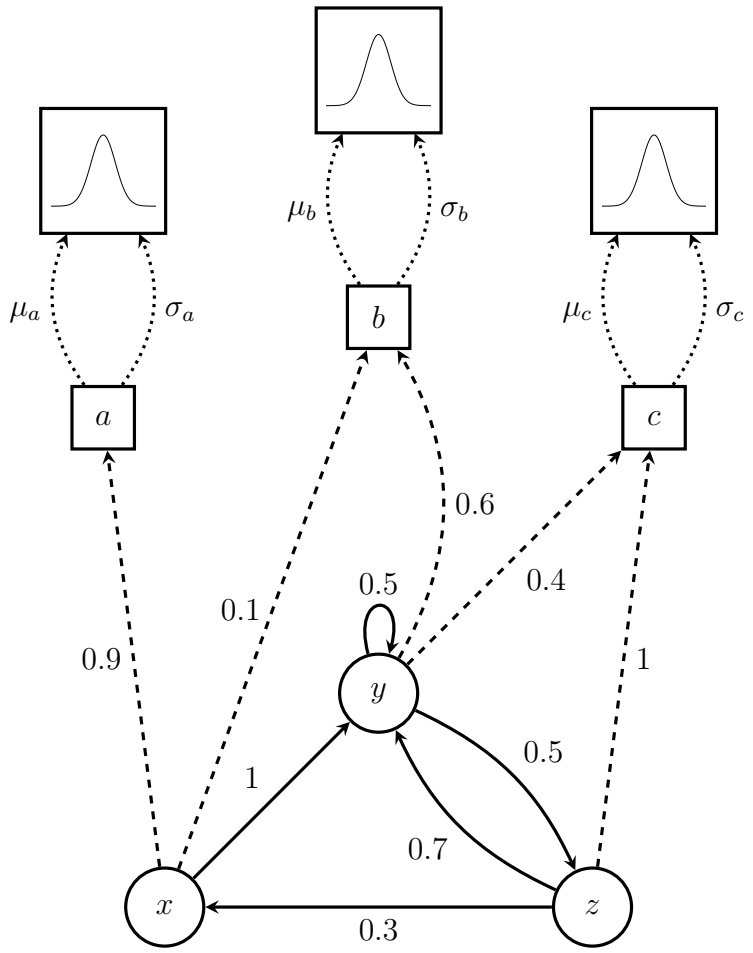


Figure 2.3: Example of a Gaussian mixture hidden Markov model, with the state set $S = \{x, y, z\}$, and sub-output set $O' = \{a, b, c\}$.

2.3 Requirements analysis

Upon successful comprehension of the background material, I refined the **success criteria** originally given in the project proposal (Appendix D) into a **statement of the primary problem** that the project intends to solve, along with a set of **deliverables** that should be produced to that end, with their respective **priorities** and mutual **dependencies**.

The statement of the primary problem is as follows:

> **Given two classes, C_1 and C_2, and two paired vectors of real-valued observations, $\vec{x}$ and $\vec{y}$ (such that x_i and y_i represent different measurements ($\sim$ *data types*) on the same entity, or on the same point in time), determine whether the pair $(\vec{x}, \vec{y})$ is a member of C_1 or C_2.**

This represents a *binary classification* problem, where the input to be classified consists of two types of data. For example, in the context of diagnostics, C_1 could refer to 'patient' and C_2 to 'normal', and $\vec{x}$ and $\vec{y}$ could refer to two types of measurements obtained from a particular subject, such that x_i and y_i were measured at the same time (e.g. from the same blood draw). This formulation has been chosen primarily with evaluation in mind, as there exists a wide variety of established metrics for evaluating binary classifiers [36].

The list of deliverables for the project is outlined in Table 2.1. Most of them are explicitly mentioned in the project proposal, with a few additions and removals.

Deliverable	**Priority**	**Difficulty**
Basic chain (GM)HMM implementation	**P0**	**Medium**
Multiplex network implementation	**P0**	**Medium**
Multiobjective optimisation algorithm	**P0**	**High**
Multiplex chain GMHMM (integrating all of the above)	**P0**	**Medium**
Evaluation suite	**P0**	**High**
Full multiplex GMHMM implementation	**P1**	**High**
Noise testing suite	**P1**	**Medium**
Multiplex network visualisation (via `muxViz` (§1.3.3))	**P2**	**Low**
Synthetic data generator (for evaluation)	**P2**	**Low**

Table 2.1: A summary of the deliverables within this project.

Deliverables with priority **P0** are needed for the **core project**—construction of a basic classifier model and evaluating its performance with the addition of multiplexity—and their proper execution represents the primary success criterion; priorities **P1** and **P2** represent the strongly desirable and desirable **extensions**, respectively.

Dependency analysis (the results of which are outlined in Figure 2.4) revealed that the three main building blocks of the project are *independent* (allowing for an inherently **modular** design); integrating them into a single model requires an interface to all three, and the evaluation suite requires an interface to the integrated structure. There were no interdependencies discovered between the deliverables with priorities **P1** and **P2**, which meant that the extensions could be implemented in any order after the core elements were in place.

2.4 Choice of tools

2.4.1 Programming languages and libraries

There was a wide variety of programming languages I could have used for accomplishing the tasks outlined above; however, to me it was clear from the onset that the language to be used should be **C++** [42], mainly for the reasons of:

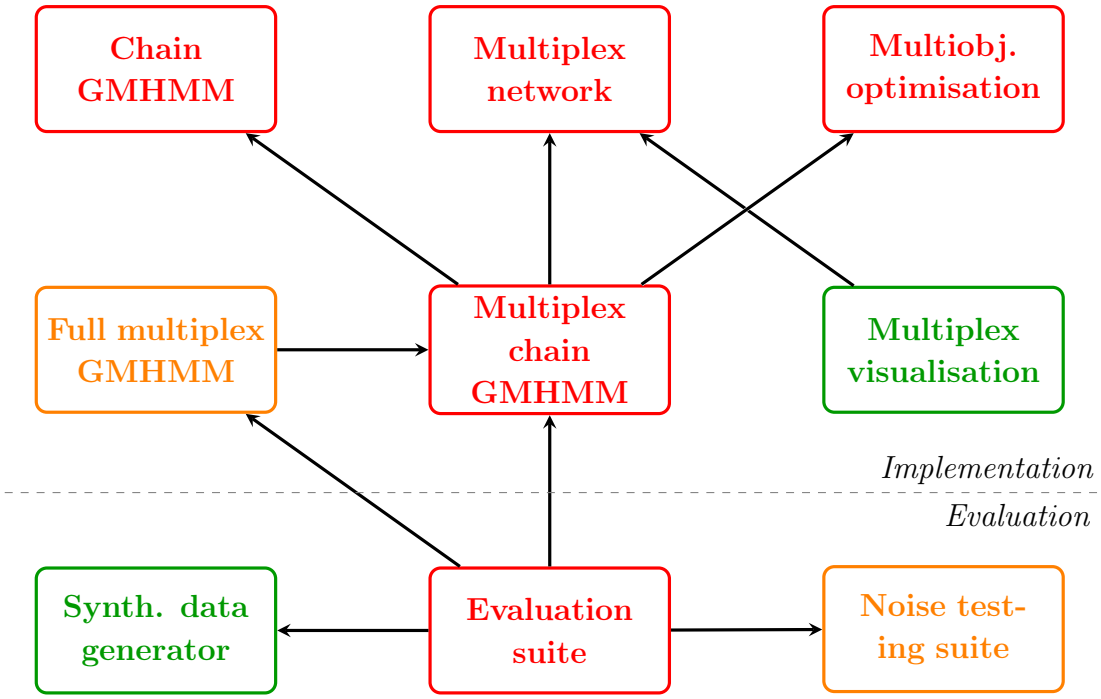


Figure 2.4: The dependencies found between the components of the project. The colour of a component represents its priority level (**High** to **Low**). A directed edge $x \longrightarrow y$ means x is dependent on y.

- *Efficiency*; I wanted to make the final product capable of efficiently handling large data sets—and the C/C++ languages were designed with performance and efficiency as their primary goals;
- *Proficiency*; At the time of proposing the project, I was already highly familiar with C++, using it mostly for the purpose of algorithmic programming competitions as well as implementing a variety of data structures and algorithms[3].
- *Standard Template Library*; C++ offers a selection of higher-level constructs and implemented data structures and algorithms, as part of the STL.

As mentioned in §1.2, the project was not expected to rely on any external libraries for the core implementation; as such, I only used the constructs available within the C++11 STL for the project implementation. For the purpose of executing the test harnesses on the components of the model, several **shell scripts** were written.

2.4.2 Development environment

The entirety of the implementation, evaluation and dissertation writeup work has been carried out on my own machine (2.6 GHz Intel Core i7 with 8 GB RAM, running Mac OS X 10.10 *Yosemite*). The full list of tools utilised for developing the project and writing up the dissertation is given in Table 2.2.

[3] https://github.com/PetarV-/Algorithms

Tool	Purpose
`vim 7.3`	Text editor
`Xcode 6.1`	IDE
`clang 3.6`	Compiler
`GNU Make 3.81`	Build automation
`git 2.3.2`	Revision control
`rsync 2.6.9`	File transfer
`cron`	Job scheduling
TEXpad 1.7.9	LaTeX editor
PGF/TikZ 3.0	Graphics package
`gnuplot 5.0`	Graphing utility

Table 2.2: Tools used for the development of the project.

2.4.3 Backup strategy

Special attention has been given to devising a resilient backup strategy for this project's codebase and dissertation sources. Using the `git` revision control system allowed me to store a backup of my repository on the hosting service *GitHub*. For redundancy, I also setup an automatic synchronisation of the project onto my personal file spaces on *Dropbox* and *Google Drive*, utilising `rsync` and `cron`. Lastly, I also made regular manual backups of my machine's entire filesystem onto an external 1TB HDD, with Apple's *Time Machine*.

The backup strategy is summarised in Figure 2.5.

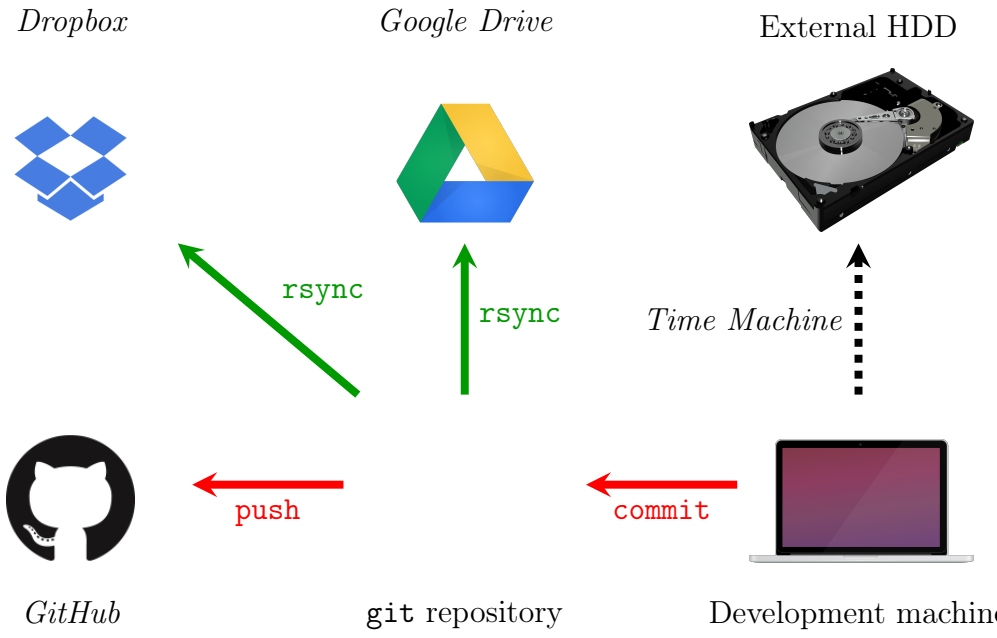


Figure 2.5: Overview of the project's enforced backup strategy.

2.5 Software engineering techniques

After reviewing several available software engineering models [31] I have decided to adopt the *Iterative Development Model* for implementing my project. This allowed for a less strict set of requirements at the onset compared to the Waterfall model (allowing for features to be included or excluded as the project was ongoing), and provided me with flexibility to implement the proposed deliverables in an isolated manner from one another, taking advantage of the modular design I proposed for this project.

Furthermore, I have adhered to good software development practices while developing the project, such as:

- Keeping the project's source tree organised into a logical subfolder structure (a screenshot of which is shown in Figure 2.6);
- Exposing a clear interface specification for each component in its `.h` (header) file (all of which were stored in a common `include/` directory);
- Treating all compiler warnings and constructs not adhering to the C++11 standard as errors; I have compiled all of the source files with the following flags:
  `CFLAGS = -std=c++11 -O3 -Wall -Wextra -Werror -Weffc++ -Wstrict-aliasing --pedantic -c`

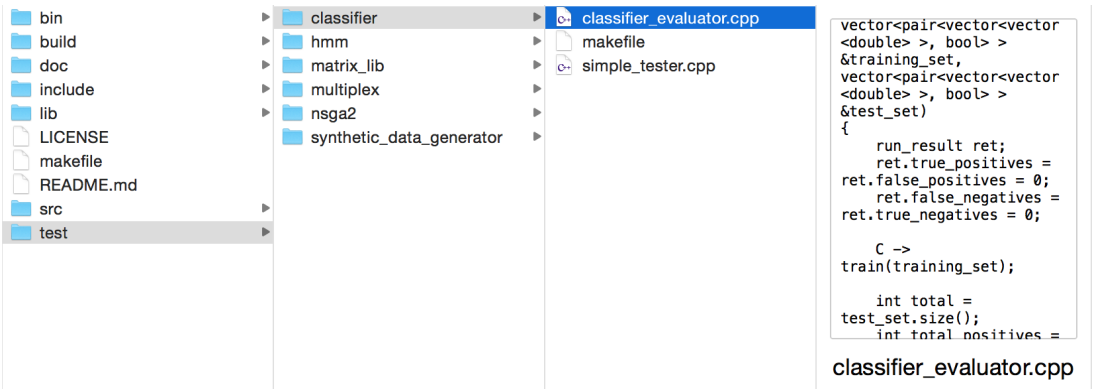


Figure 2.6: Overview of the project's source tree.

2.6 Summary

This chapter has summarised the work I completed before the implementation phase of the project began; in particular, I have provided a review of the background theory underpinning the project (specifically, multiplex networks and hidden Markov models), and discussed the analysis of the project's requirements and deliverables, as well as the relevant tools and methodologies adopted.

The next chapter will focus on the implementation of the project, giving details on how the proposed goals were actually achieved.

Chapter 3

Implementation

In this chapter, I will thoroughly describe the implementation phase of my project, resulting in successfully achieving the deliverables outlined in the Preparation chapter. This will follow the logical order of the implementation strategy adopted for the project—starting from the core project components, following up with a discussion of the extensions.

The final implementation consists of over 5000 lines of C++, and as such it is not suitable to showcase large code excerpts; rather, this chapter will focus on the high-level descriptions of the implemented algorithms (using pseudocode where appropriate), while larger code fragments will be confined to the appendices (ref. Appendix B.)

3.1 Overview

The sections of the Implementation chapter will discuss the achievement of the deliverables related to the various models of machine learning that are capable of taking advantage of multiplexity, as outlined in §2.3. Initially, a brief overview of the *supervised learning setup* (which is common to all of the implemented models) will be given in §3.2; afterwards, the implementation of the models will be discussed in the following order:

1. **Multiplex chain GMHMM** (§3.3): the simpler of the models, implementing a highly simplified GMHMM layer as its basis, to allow for easier implementation, suitable for the core deliverable. Its key components will be described in turn (§3.3.1–3.3.3).

2. **Full multiplex GMHMM** (§3.4): once the core model is properly discussed, I will describe how to generalise it to utilise a completely flexible GMHMM in each layer of the multiplex network. As the model has many similarities to the chain version, only the most major difference (the training algorithm) will be discussed thoroughly (§3.4.2–3.4.3).

3.2 Supervised learning setup

All of the implemented machine learning models will be solving a *supervised learning* [29] problem. This specifies the basic top-level interface that each model must expose, and hence is a logical place to initiate the discussion of the implementation phase.

A model for the supervised learning problem aims to construct a *labelling function*, $h : X \to Y$, to assign labels from a given set Y to inputs from a set X. This function is constructed as the output of a *learning algorithm*, $L : (X \times Y)^n \to (X \to Y)$, which attempts to generalise from a set of inputs with known labels—a *training set*—to produce a function capable of labelling previously *unseen* inputs. The process is summarised by the diagram given in Figure 3.1.

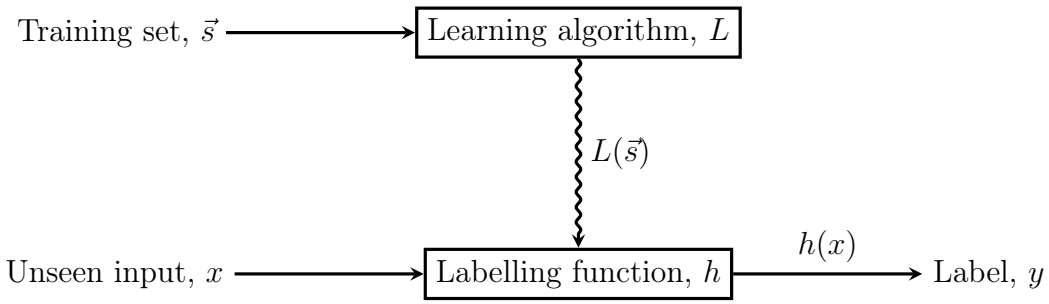


Figure 3.1: A diagram of the supervised learning setup.

As such, each of the models we consider has to provide an implementation of at least these two functions, `train(`$\overrightarrow{(X,Y)}$`)` and `classify(`X`)`.

The statement of the primary problem (given in § 2.3) specifies that $Y = \{C_1, C_2\}$ (binary classification), and also that $X = (\mathbb{R} \times \mathbb{R})^n$. However, for making the interface more easily extendable to other kinds of problems, it has been templated with the types corresponding to the sets X and Y, as described in Listing 3.1; a typical model discussed within this project will extend from `Classifier<vector<vector<double> >, bool>`[1].

```
template<typename Data, typename Label>
class Classifier
{
public:
  virtual ~Classifier() { }
  virtual void train(std::vector<std::pair<Data, Label> > &training_set)
    = 0;
  virtual Label classify(Data &test_data) = 0;
};
```

Listing 3.1: An interface for a classifier in the supervised learning setup.

[1] **N.B.** a `vector` was used for the second dimension rather than `pair`, to easily allow for using the structure with more than two types of data at once.

3.3 Multiplex chain GMHMM

3.3.1 Chain GMHMM

The simplest of the single-layer models considered is a GMHMM where all of the states are arranged in a *chain-like structure* (as depicted in Figure 3.2). The model always starts in state 0, with each state transitioning into the next one along the chain with probability 1. Each state on the chain has its own probability distribution of the sub-outputs (represented in the form of the usual *emission probability matrix*, $\mathbf{O}'$), and each sub-output stores its own parameters for the output's Gaussian distribution (μ, σ).

It is assumed that the amount of states in the chain is **equal** to the amount of outputs observed, that it is known which sub-output produced each output, and that the outputs can be ordered in some way; if they represent temporal data, then the ordering is trivial (sorted by time). Otherwise, a procedure should be first employed to sort the outputs in some sensible way (e.g. [30]).

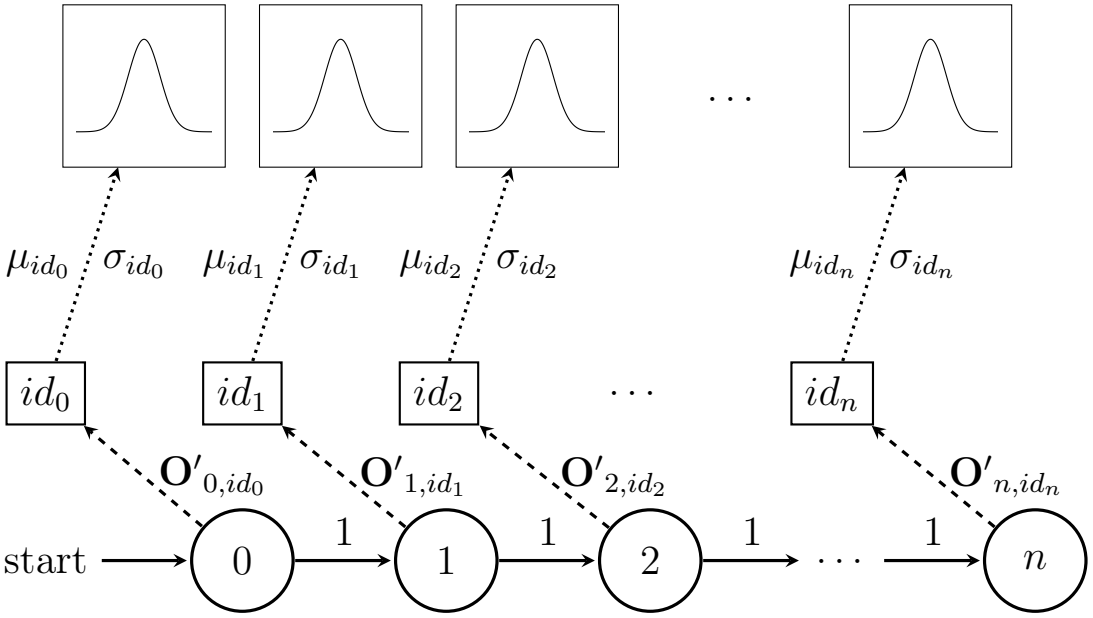


Figure 3.2: An example "trace of execution" of a chain GMHMM with $n+1$ states, with a sub-output sequence $\{id_i\}_{i=0}^{n}$.

This is a very "lightweight" model, in the sense of the amount of data that needs to be stored to maintain it—as shown by Listing 3.2, only the amount of states and sub-outputs, the matrix $\mathbf{O}'$ and the parameters $\vec{\mu}$ and $\vec{\sigma}$ need to be maintained.

3.3.1.1 Output likelihood

The main building block for the inference algorithm for the chain GMHMM (and all of the other models that will be considered) is the method that determines the **likelihood**, $\mathbb{L}(\vec{y}, \vec{id})$, of a given output sequence being produced by the model.

```
class SimpleChainGMHMM
{
private:
  int obs, sub_obs; // number of states and "sub-observations"
  double **G; // sub-observation emission probabilites
  double *mu, *sigma; // means and variances for each sub-observation

public:
  ... // methods
};
```

Listing 3.2: The state of a chain GMHMM.

Note the change in terminology; we are looking for the *likelihood* rather than the *probability* of the sequence being emitted. This is due to the fact that, in continuous space, the probability of any individual sequence being observed is zero. In the context of likelihood, we will use the *Gaussian probability density function*, $\mathcal{N}(x; \mu, \sigma)$, instead of the actual probability, because what we are interested in is typically **not** the actual likelihood, but rather *how it compares to other likelihoods*.

For a chain GMHMM $\mathbf{\Theta}$, the output likelihood is easily expressible as

$$\mathbb{L}(\vec{y}, \vec{id}|\mathbf{\Theta}) = \prod_{i=0}^{n} \mathbf{O'}_{i,id_i} \mathcal{N}(y_i; \mu_{id_i}, \sigma_{id_i})$$

i.e. the product of the probabilities of emitting each sub-output, multiplied with the likelihoods of emitting each output from its sub-output. However, a product of many such floating-point values, particularly if several of them are close to zero, is prone to underflow. As such, it is typically better to calculate the **log-likelihood** instead:

$$\log\left(\mathbb{L}(\vec{y}, \vec{id}|\mathbf{\Theta})\right) = \sum_{i=0}^{n} \left(\log\left(\mathbf{O'}_{i,id_i}\right) + \log\left(\mathcal{N}(y_i; \mu_{id_i}, \sigma_{id_i})\right)\right)$$

This formula lends itself to a simple $O(n)$-time algorithm which simply sums up the relevant values.

3.3.1.2 Parameter estimation

Training a chain GMHMM consists of estimating its state $(\mathbf{O'}, \vec{\mu}, \vec{\sigma})$ from a set of output sequences presumed to be generated by the model. The implication of this is that the state should be chosen such that the likelihoods of the sequences being produced by the model are high, while still allowing for flexibility (i.e. not *overfitting* the model).

The parameters $\vec{\mu}$ and $\vec{\sigma}$ are easy to estimate; for a given sub-output x, we extract all of the k given outputs produced by it, $\vec{o_x}$, and then use their *sample mean* and *corrected sample standard deviation* to estimate μ_x and σ_x:

$$\mu_x = \frac{1}{k} \sum_{i=1}^{k} (o_x)_i$$

$$\sigma_x = \sqrt{\frac{1}{k-1}\sum_{i=1}^{k}((o_x)_i - \mu_x)^2}$$

The estimation of $\mathbf{O}'$ is less straightforward, and a variety of schemes could be used. I devised a method that assigns to each field of the matrix a **score**, with the scores being normalised for each row of the matrix in the end. A naïve scheme would simply increment the score of $\mathbf{O}'_{xy'}$ whenever the state x produces the sub-output y' in the training set, however this can lead to some suitable sequences being evaluated with the likelihood of zero—if they were to use an entry of the matrix that was never seen in the training data.

To compensate for this, I devised a metric that ensures the following three features:

- The naïve scheme's increments are still present; whenever state x produces sub-output y' in the training data, $\mathbf{O}'_{xy'} \leftarrow \mathbf{O}'_{xy'} + 1$.
- For each training sequence, each field of the matrix $\mathbf{O}'$ must be increased by **at least** a given small, but manageable real number ε; within my project I set $\varepsilon = 10^{-6}$.
- For a given training sequence $s = (\vec{y}, \vec{id})$, a particular field $\mathbf{O}'_{xy'}$'s assigned score should **decay exponentially** as $d_s(id_x, y')$, the *distance* between y' and the sub-output actually produced by x (id_x), increases.

The choice of $d_s(id_x, y')$ is left to the user; a simple definition would be to take the smallest possible $|x - x'|$, where x' is a state producing y' in the given training sequence, along with a fixed value (perhaps $n + 1$) if there is no such x' in the sequence.

With respect to the given features, a possible scoring function is

$$score_s(x, y') = e^{-\lambda d_s(id_x, y')}$$

where $\lambda = \frac{-\ln \varepsilon}{M}$, and M is the *maximal* observed value of $d_s(id_x, y')$ in the training set. A plot of this function is given in Figure 3.3; it clearly satisfies all of the desired properties.

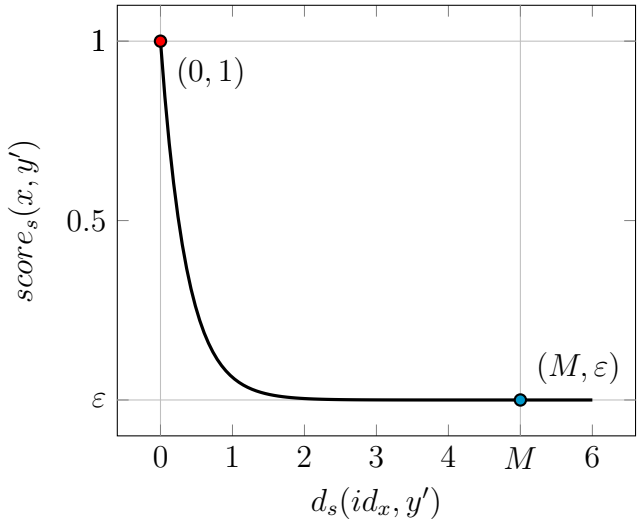


Figure 3.3: A plot of $score_s(x, y')$ as a function of $d_s(id_x, y')$, given $M = 5$.

Taking into account all of the above, Algorithm 1 represents the full training algorithm for a chain GMHMM. This algorithm has an asymptotic time complexity of $O(n \cdot m \cdot s)$ (dominated by the triply-nested `for` loops), where n is the number of states, m is the number of sub-outputs, and s is the number of training sequences in the set $\vec{s}$.

3.3.1.3 Classifier construction

Once the training procedure is properly defined, all of the elements needed for constructing a binary classifier using the chain GMHMM are in place:

`train` This binary classifier, C_{chain}, will encapsulate **two** chain GMHMMs, $\mathbf{\Theta_1}$ and $\mathbf{\Theta_2}$, one to be constructed using all the training sequences assigned to C_1, the other trained on all the sequences assigned to C_2 (Algorithm 2).

`classify` Classifying an unseen sequence reduces to evaluating the (log-)**likelihoods** of the sequence being produced by *each* of the two chain GMHMMs, choosing the class corresponding with the *larger* likelihood (Algorithm 3).

The same methodology will also be utilised in the other models discussed; as such, to define a new classifier from any model, it will be sufficient to specify the *training* and *likelihood* functions for the model.

Algorithm 2 Chain GMHMM binary classifier: Training

Input: Binary classifier $C_{chain} = (\mathbf{\Theta_1}, \mathbf{\Theta_2})$, two training sets of sequences $\vec{s_1}$ and $\vec{s_2}$, labelled with C_1 and C_2 respectively.

1: **procedure** TRAIN(C_{chain}, $\vec{s_1}$, $\vec{s_2}$) ▷ Train the classifier using $\vec{s_1}$ and $\vec{s_2}$
2: TRAINCHN($\mathbf{\Theta_1}$, $\vec{s_1}$, d, 10^{-6}) ▷ Train the first model using $\vec{s_1}$
3: TRAINCHN($\mathbf{\Theta_2}$, $\vec{s_2}$, d, 10^{-6}) ▷ Train the second model using $\vec{s_2}$
4: **end procedure**

Algorithm 3 Chain GMHMM binary classifier: Classification

Input: Binary classifier $C_{chain} = (\mathbf{\Theta_1}, \mathbf{\Theta_2})$, sequence $(\vec{y}, \vec{id})$ to be classified.

1: **function** CLASSIFY(C_{chain}, $\vec{y}$, $\vec{id}$) ▷ Classify $\vec{y}$ using C_{chain}
2: $\ell_1 \leftarrow$ LOGLIKELIHOOD($\mathbf{\Theta_1}$, $\vec{y}$, $\vec{id}$) ▷ Get likelihood of $(\vec{y}, \vec{id})$ produced by $\mathbf{\Theta_1}$
3: $\ell_2 \leftarrow$ LOGLIKELIHOOD($\mathbf{\Theta_2}$, $\vec{y}$, $\vec{id}$) ▷ Get likelihood of $(\vec{y}, \vec{id})$ produced by $\mathbf{\Theta_2}$
4: **if** $\ell_1 > \ell_2$ **then return** C_1 ▷ Return the class according to the likelihoods
5: **else return** C_2
6: **end if**
7: **end function**

Output: The class that $(\vec{y}, \vec{id})$ belongs to, according to C_{chain}.

3.3.2 Combining chains

Now that the structure for classifying on a single type of data is fully specified, I will show how to extend it to a multiplex classifier operating on several data types simultaneously.

Algorithm 1 Chain GMHMM parameter estimation

Input: Chain GMHMM $\mathbf{\Theta} = (n, m, \mathbf{O}', \vec{\mu}, \vec{\sigma})$, training set $\vec{s} = \overrightarrow{(\vec{y}, \vec{id})}$, distance function d, minimal score gain ε.

1: **procedure** TRAINCHN($\mathbf{\Theta}$, $\vec{s}$, d, ε) ▷ Estimate the parameters of $\mathbf{\Theta}$ using $\vec{s}$
2: **for** $y' \leftarrow 0$ **to** m **do**
3: $\vec{o_{y'}} \leftarrow \emptyset$ ▷ Initialise the sets of outputs for each sub-output
4: $k_{y'} \leftarrow 0$
5: **end for**
6: $M \leftarrow 0$ ▷ Initialise maximal distance observed
7: **for all** $s = (\vec{y}, \vec{id}) \in \vec{s}$ **do**
8: **for** $x \leftarrow 0$ **to** n **do**
9: **for** $y' \leftarrow 0$ **to** m **do**
10: $M \leftarrow \max(M, d_s(id_x, y'))$ ▷ Update maximal distance
11: **end for**
12: $\vec{o_{id_x}}$.PUSHBACK(y_x) ▷ Fill the sets of outputs
13: $k_{id_x} \leftarrow k_{id_x} + 1$
14: **end for**
15: **end for**
16: **for** $y' \leftarrow 0$ **to** m **do** ▷ Estimate $\vec{\mu}$ and $\vec{\sigma}$
17: ASSERT($k_{y'} > 1$) ▷ *Precondition:* Must have at least two outputs
18: $\mu_{y'} \leftarrow \frac{1}{k_{y'}} \sum_{i=1}^{k_{y'}} (o_{y'})_i$
19: $\sigma_{y'} \leftarrow \sqrt{\frac{1}{k_{y'}-1} \sum_{i=1}^{k_{y'}} ((o_{y'})_i - \mu_{y'})^2}$
20: **end for**
21: $\lambda \leftarrow \frac{-\ln \varepsilon}{M}$ ▷ Set the λ parameter used for $score_s$
22: $\mathbf{O}' \leftarrow \mathbf{0}_{(n+1)\times(m+1)}$ ▷ Set the entries of $\mathbf{O}'$ to zero
23: **for all** $s = (\vec{y}, \vec{id}) \in \vec{s}$ **do**
24: **for** $x \leftarrow 0$ **to** n **do**
25: **for** $y' \leftarrow 0$ **to** m **do**
26: $\mathbf{O}'_{xy'} \leftarrow \mathbf{O}'_{xy'} + score_s(x, y')$ ▷ Assign score to each entry
27: **end for**
28: **end for**
29: **end for**
30: **for** $x \leftarrow 0$ **to** n **do**
31: $S \leftarrow \sum_{y'=0}^{m} \mathbf{O}'_{xy'}$
32: **for** $y' \leftarrow 0$ **to** m **do**
33: $\mathbf{O}'_{xy'} \leftarrow \frac{\mathbf{O}'_{xy'}}{S}$ ▷ Normalise each row of the matrix
34: **end for**
35: **end for**
36: **end procedure**

In this subsection I will outline the parameters needed to specify the model, and give an algorithm for determining likelihoods of sequences being produced by it.

3.3.2.1 Multiplex chain

The chain GMHMM structure described thus far is very simple and not really a "hidden" Markov model, as the sequence of states it will take is *pre-determined* by the parameters of the chain. This has been done to make *combining* several chains as simple as possible.

Assuming we have L types of data in our output sequences, we may train L separate chain GMHMMs; one on each of the types. Then, when producing any given sequence, we start in one of the chains, but *allow the current chain to change* as we go along. This essentially means that the model will traverse along a chain as usual, but possibly change the type of data being used to produce the current full data point.

The full structure should still represent an HMM, but now the starting state is no longer pre-determined; there are L states responsible for processing the first data point. This requires the specification of a *start-state probability vector*, $\vec{\pi}$, determining the probabilities of starting in each of the chains. Furthermore, we need to specify the probabilities of transitioning between each pair of chains; this may be conveniently represented as a matrix $\boldsymbol{\omega}$ of size $L \times L$, such that $\boldsymbol{\omega}_{ij}$ represents the probability of transitioning from chain i to chain j (or staying in chain i if $i = j$; in fact, we may omit $\vec{\pi}$ if we set π_i to be equal to the normalised entries of $\boldsymbol{\omega}_{ii}$). Observing the entire structure as an HMM, there is a transition from state x in layer i to state $x + 1$ in layer j with probability $\boldsymbol{\omega}_{ij}$. An example of a two-layer model is given in Figure 3.4[2].

Most of the necessary information for specifying this model is already contained within the individual chains; as shown by Listing 3.3, along with the chains themselves, only the transition probability matrix, $\boldsymbol{\omega}$, and the amount of chains in the model, L, need to be stored (we may also store the number of states and sub-observations, for convenience).

3.3.2.2 Forward algorithm

As the model can now be observed as a general hidden Markov model (not being known exactly which state sequence the model will follow), a new method for calculating likelihoods is needed; I will first introduce the solution for a generalised GMHMM, and then briefly discuss ways in which it could be optimised for the chain GMHMM.

Given a GMHMM $\boldsymbol{\Theta} = (n, m, \vec{\pi}, \mathbf{T}, \mathbf{O}', \vec{\mu}, \vec{\sigma})$, we may attempt to evaluate the likelihood of an output sequence similarly as in § 3.3.1.1; assuming we have an output sequence $(\vec{y}, \vec{id})$ of length T, the total likelihood of the sequence being produced by the

[2] **N.B.** this model does not fit the definition of a multiplex network as specified in §2.1.3, as the network is not *diagonally coupled*; however, it is possible to transform it into an equivalent multiplex network by *adding more layers*.

```
class HMMChainMultiplex
{
private:
  int obs, sub_obs; // as before
  int L; // number of layers (data-types)
  std::vector<SimpleChainGMHMM*> layers; // chains
  double **omega; // transition probabilities
public:
  ... // methods
};
```

Listing 3.3: The state of a multiplex chain GMHMM.

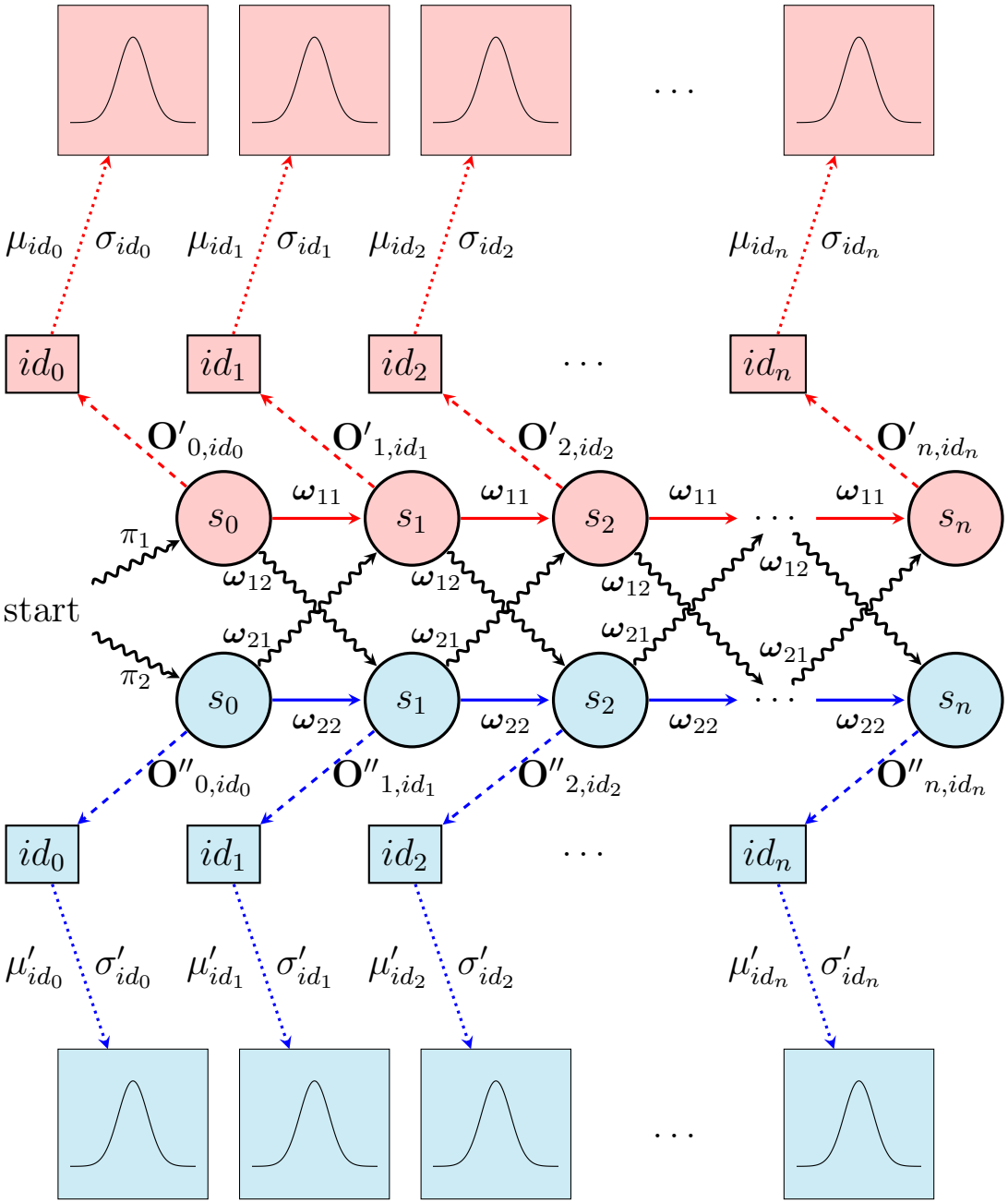


Figure 3.4: Example of a multiplex chain GMHMM with two layers.

GMHMM may be evaluated by summing over all possible state sequences:

$$\mathbb{L}(\vec{y}, \vec{id}|\mathbf{\Theta}) = \sum_{\vec{x} \in S^T} \mathbb{L}(\vec{y}, \vec{id}|\vec{x}, \mathbf{\Theta})\mathbb{L}(\vec{x}|\mathbf{\Theta})$$

However, the number of possible state sequences is *exponential* in T; hence this approach is infeasible. For a better approach, note that a lot of the calculations performed this way are redundant, and reusing previously calculated results can speed up the evaluation; this leads to a *dynamic programming* algorithm known as the **forward algorithm**.

The forward algorithm computes **forward likelihoods**, $\alpha_t(x)$, corresponding to the likelihood of processing the first t elements of the output sequence and ending up in state x:

$$\alpha_t(x) \stackrel{\text{def}}{=} \mathbb{L}(\{(y_i, id_i)\}_{i=1}^t, x_t = x|\mathbf{\Theta})$$

Having computed all of the values of $\alpha_t(x)$, the overall likelihood may be evaluated as

$$\mathbb{L}(\vec{y}, \vec{id}|\mathbf{\Theta}) = \sum_{x \in S} \mathbb{L}(\vec{y}, \vec{id}, x_T = x|\mathbf{\Theta}) = \sum_{x \in S} \alpha_T(x)$$

The *base cases*, $\alpha_1(x)$, can be simply computed as

$$\alpha_1(x) = \mathbb{L}((y_1, id_1), x_1 = x|\mathbf{\Theta}) = \pi_x \mathbf{O'}_{x,id_1} \mathcal{N}(y_1; \mu_{id_1}, \sigma_{id_1})$$

Assuming we have already computed $\alpha_t(x)$ for all x, we can derive an expression for $\alpha_{t+1}(x)$ as follows (the dependencies between the previously calculated likelihoods and the new one are illustrated in Figure 3.5):

$$\begin{aligned}
\alpha_{t+1}(x) &= \mathbb{L}(\{(y_i, id_i)\}_{i=1}^{t+1}, x_{t+1} = x|\mathbf{\Theta}) \\
&= \mathbb{L}((y_{t+1}, id_{t+1})|x_{t+1} = x, \{(y_i, id_i)\}_{i=1}^t, \mathbf{\Theta})\mathbb{L}(\{(y_i, id_i)\}_{i=1}^t, x_{t+1} = x|\mathbf{\Theta}) \\
&= \mathbb{L}((y_{t+1}, id_{t+1})|x_{t+1} = x, \mathbf{\Theta}) \sum_{x' \in S} \mathbb{L}(\{(y_i, id_i)\}_{i=1}^t, x_{t+1} = x, x_t = x'|\mathbf{\Theta}) \\
&= \mathbf{O'}_{x,id_{t+1}} \mathcal{N}(y_{t+1}; \mu_{id_{t+1}}, \sigma_{id_{t+1}}) \sum_{x' \in S} \mathbb{L}(\{(y_i, id_i)\}_{i=1}^t, x_t = x'|\mathbf{\Theta})\mathbb{L}(x_{t+1} = x|x_t = x', \mathbf{\Theta}) \\
&= \mathbf{O'}_{x,id_{t+1}} \mathcal{N}(y_{t+1}; \mu_{id_{t+1}}, \sigma_{id_{t+1}}) \sum_{x' \in S} \alpha_t(x')\mathbf{T}_{x'x}
\end{aligned}$$

This requires only $O(n)$ steps per state-time pair, and as such the overall time complexity of the forward algorithm is $O(n^2 \cdot T)$.

The forward algorithm is prone to the same underflow problems as the algorithm given in §3.3.1.1; however, as we are *adding* components together instead of multiplying them, it is no longer suitable to trivially take the logarithm. A common solution is to normalise the entries of $\alpha_t(x)$ for each t, with a **scaling coefficient** c_t:

$$c_t = \frac{1}{\sum\limits_{x \in S} \alpha_t(x)}$$

The actual value of $\alpha_t(x)$ stored for future steps is

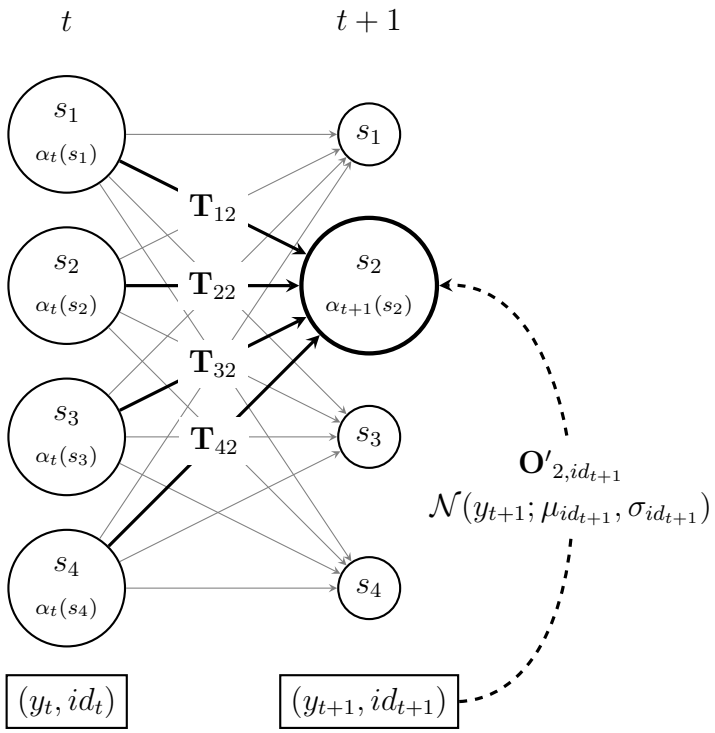


Figure 3.5: Illustration of the recurrence relation of the forward algorithm on a GMHMM with four states; displaying all the variables that $\alpha_{t+1}(s_2)$ is dependent on.

$$\hat{\alpha}_t(x) = c_t \alpha_t(x)$$

As the scaling coefficients propagate throughout the further calculations, the full likelihood of the sequence may be calculated as

$$\mathbb{L}(\vec{y}, \vec{id}|\mathbf{\Theta}) = \frac{\sum_{x \in S} \hat{\alpha}_T(x)}{c_1 c_2 \cdots c_T} = \prod_{t=1}^{T} \frac{1}{c_t}$$

the logarithm of which is easily maintainable as

$$\log \mathbb{L}(\vec{y}, \vec{id}|\mathbf{\Theta}) = -\sum_{t=1}^{T} \log c_t$$

The pseudocode of the forward algorithm is given within Algorithm 4. It could be optimised for the multiplex chain GMHMM by taking advantage of the fact that at time t we may only be in state t, in one of the L chains; as such, it suffices to store $\alpha_t(l)$, where l ranges *over all chains*; this reduces the time complexity to $O(L^2 \cdot T)$.

Algorithm 4 Forward algorithm

Input: GMHMM $\mathbf{\Theta} = (n, m, \vec{\pi}, \mathbf{T}, \mathbf{O}', \vec{\mu}, \vec{\sigma})$, sequence $(\vec{y}, \vec{id})$ of length T.

function FORWARD($\mathbf{\Theta}$, $\vec{y}$, $\vec{id}$, T) ▷ Evaluate the likelihood $\mathbb{L}(\vec{y}, \vec{id}|\mathbf{\Theta})$
  for $x \leftarrow 1$ **to** n **do**
    $\alpha_1(x) \leftarrow \pi_x \mathbf{O'}_{x,id_1} \mathcal{N}(y_1; \mu_{id_1}, \sigma_{id_1})$ ▷ Base cases; $\alpha_1(x)$
  end for
  $c_1 \leftarrow \frac{1}{\sum_{x=1}^{n} \alpha_1(x)}$ ▷ Calculate the scaling coefficient
  for $x \leftarrow 1$ **to** n **do**
    $\alpha_1(x) \leftarrow c_1 \alpha_1(x)$ ▷ Normalise the entries
  end for
  for $t \leftarrow 2$ **to** T **do**
    for $x \leftarrow 1$ **to** n **do** ▷ Recurrence relation for $\alpha_t(x)$
      $\alpha_t(x) \leftarrow \mathbf{O'}_{x,id_t} \mathcal{N}(y_t; \mu_{id_t}, \sigma_{id_t}) \sum_{x'=1}^{n} \alpha_{t-1}(x') \mathbf{T}_{x'x}$
    end for
    $c_t \leftarrow \frac{1}{\sum_{x=1}^{n} \alpha_t(x)}$ ▷ Calculate the scaling coefficient
    for $x \leftarrow 1$ **to** n **do**
      $\alpha_t(x) \leftarrow c_t \alpha_t(x)$ ▷ Normalise the entries
    end for
  end for
  $\ell \leftarrow -\sum_{t=1}^{n} \log c_t$ ▷ Calculate the log-likelihood
  return $(\alpha, \vec{c}, \ell)$
end function

Output: The forward likelihoods $\alpha_t(x)$, the scaling coefficients $\vec{c}$, the log-likelihood ℓ.

3.3.3 Multiplex training

With an algorithm for evaluating likelihoods in place, the only remaining component that needs to be specified for the multiplex chain GMHMM classifier to be constructed is the *training algorithm*. As previously specified, the individual chains within the multiplex may be trained with the parameter estimation method defined in §3.3.1.2; once this is done, estimating the optimal interlayer transition probabilities ($\boldsymbol{\omega}$) remains. In this subsection I will introduce the optimisation algorithm that was chosen for this purpose; afterwards I will briefly discuss the integration of this algorithm with the multiplex network structure, in order to round off the training algorithm for the core model.

3.3.3.1 Multiobjective optimisation

As mentioned in §3.3.1.2, the model parameters we are aiming to obtain upon training should optimise the likelihoods of the sequences within the training set being produced, while still maintaining a level of flexibility.

At the most essential level, we would like to optimise all of the likelihoods *independently* of one another; this corresponds to a **multiobjective optimisation problem**; for a given training set of k sequences, $\vec{s}$, we would like to maximise, for a given (partially-trained) multiplex chain GMHMM $\boldsymbol{\Theta}$, the sequence of functions $\{f_i(\boldsymbol{\omega})\}_{i=1}^{k}$, defined as the likelihoods of each of the sequences in the training set being produced by the model, assuming the interlayer transition probabilities are $\boldsymbol{\omega}$, i.e.

$$f_i(\boldsymbol{\omega}) = \mathbb{L}(s_i|\boldsymbol{\Theta}, \boldsymbol{\omega})$$

In the typical case, it is unlikely that all of the functions will reach their global maxima simultaneously at the same $\boldsymbol{\omega}$; rather, a *frontier* (the **Pareto front**) will be reached, consisting of a set of points where it is impossible to further optimise one function without degrading at least one of the others—refer to Figure 3.6. Formally, we say that a solution $\boldsymbol{\omega}$ **dominates** solution $\boldsymbol{\omega}'$, denoted $\boldsymbol{\omega} \succ \boldsymbol{\omega}'$, if

1. $\forall i.\ f_i(\boldsymbol{\omega}) \geq f_i(\boldsymbol{\omega}')$
2. $\exists j.\ f_j(\boldsymbol{\omega}) > f_j(\boldsymbol{\omega}')$

A solution $\boldsymbol{\omega}^*$ is **on the Pareto front** if there is *no other solution that dominates it.* Within the context of multiobjective optimisation, all of the solutions on the Pareto front are treated as *equally satisfactory*, and the aim of multiobjective optimisation algorithms is typically to extract as many points on the front as possible (preferably well-spread). The following part will discuss one such algorithm, which has been utilised for my project.

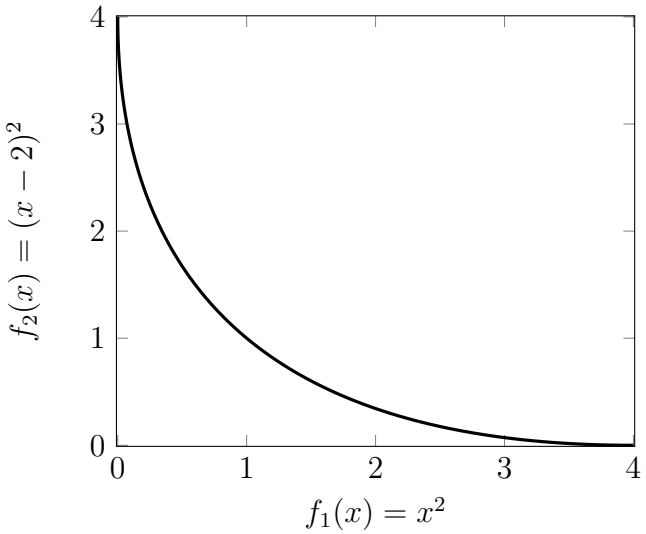


Figure 3.6: The Pareto front for minimising the functions $f_1(x) = x^2$ and $f_2(x) = (x-2)^2$; obtained when $x \in [0, 2]$.

Arguably, we could find a way to combine the likelihoods in a single function to be optimised (making the approach simpler and only extracting a single point), however, vital information about the model is potentially lost this way; in my project I have opted for the multiobjective approach to allow the user to manually discriminate which points on the frontier are more favourable, using a method of their own choosing.

3.3.3.2 NSGA-II

A popular class of algorithms for solving the multiobjective optimisation problem are *evolutionary algorithms*—population-based heuristics inspired by evolutionary concepts such as natural selection [3]. They are a good choice for a generalised optimisation problem as they *make no assumptions* about the solution space. The algorithms usually operate by *generating* a random initial population of solutions, *selecting* the best-fit group of solutions, and then *combining* them in some way, producing a set of new solutions to replace the less fit ones. The process of selecting and combining is *repeated* until the desired level of convergence is achieved.

One such algorithm is the **nondominated sorting genetic algorithm II** (*NSGA-II*) by Deb *et al.* [15]. It is a popular option because it usually provides a larger spread of solutions on the Pareto front and converges faster compared to similar evolutionary algorithms. Therefore, I decided to implement it for my project; this paragraph provides a summary of the algorithm.

The algorithm starts by randomly generating a set of n solutions, P_0, and combining them (in ways that will be discussed ahead) to obtain a set of n additional solutions, Q_0. Assuming we have a set of solutions $R_t = P_t \cup Q_t$, one iteration of NSGA-II aims to produce from it a set of more fit solutions, R_{t+1}. The first stage of the iteration is *selection*; selecting the n most fit solutions to represent the set P_{t+1}.

The primary criterion for fitness is given by *nondominated sorting*, i.e. sorting solutions by the $\succ$ ordering. This can be efficiently performed by, for each solution $p \in R_t$, keeping track of the set of solutions it dominates, S_p, and number of solutions dominating it, n_p. All solutions p with $n_p = 0$ constitute the *first nondominated front*, $\mathcal{F}_1$ and will be added to P_{t+1} first. Whenever a solution p is added to a nondominated front $\mathcal{F}_i$, for every $q \in S_p$, n_q is decremented; if $n_q = 0$, q will be a member of the next nondominated front ($\mathcal{F}_{i+1}$). This procedure is repeated until every solution has been assigned to a front.

The pseudocode is summarised in Algorithm 5, and a visualisation of its output is given in Figure 3.7. Each pair of solutions is compared by $\succ$ exactly once, and a single such comparison takes $O(m)$ steps, where m is the number of functions to optimise. After this phase, each pair of points is considered at most once while determining the nondominated fronts. As such, the initial comparisons dominate the running time of the algorithm, and hence its overall time complexity is $O(m \cdot n^2)$. The algorithm's space complexity is $O(n^2)$ (for storing the sets S_p).

Algorithm 5 Fast nondominated sorting

Input: A set of solutions, P, to sort.

function FastNondominatedSort(P) ▷ Sort the elements of P by $\succ$
  $\vec{\mathcal{F}} \leftarrow \emptyset$ ▷ Initialise the nondominated fronts
  for all $p \in P$ **do**
    $S_p \leftarrow \emptyset$ ▷ Initialise S_p and n_p
    $n_p \leftarrow 0$
    for all $q \in P$ **do**
      if $p \succ q$ **then** $S_p \leftarrow S_p \cup \{q\}$ ▷ Update S_p or n_p
      else if $p \prec q$ **then** $n_p \leftarrow n_p + 1$
      end if
    end for
    if $n_p = 0$ **then** $\mathcal{F}_1 \leftarrow \mathcal{F}_1 \cup \{p\}$ ▷ If no solution dominates p, p is on $\mathcal{F}_1$
    end if
  end for
  $i \leftarrow 1$
  while $\mathcal{F}_i \neq \emptyset$ **do** ▷ Repeat until every solution has been assigned to a front
    for all $p \in \mathcal{F}_i$ **do**
      for all $q \in S_p$ **do**
        $n_q \leftarrow n_q - 1$ ▷ q has one solution less dominating it
        if $n_q = 0$ **then** $\mathcal{F}_{i+1} \leftarrow \mathcal{F}_{i+1} \cup \{q\}$
        end if
      end for
    end for
    $i \leftarrow i + 1$
  end while
  return $\vec{\mathcal{F}}$
end function

Output: The nondominated fronts of P, $\vec{\mathcal{F}}$.

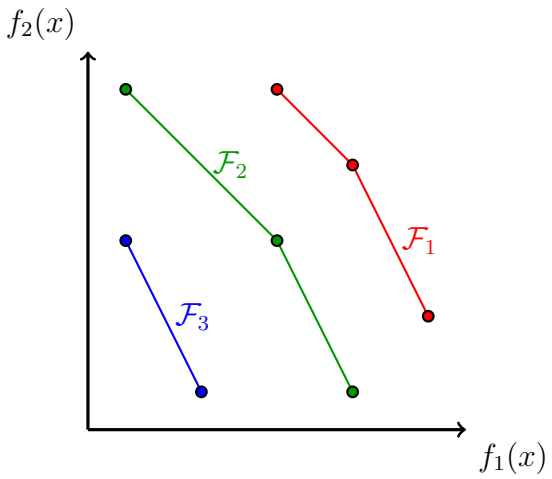


Figure 3.7: Example output of nondominated sorting; lines are drawn for clarity.

Usually, the final front that is to be added to P_{t+1} cannot fit entirely; as such, we must further discriminate the solutions within that front. In this stage, the *diversity* of the solutions is considered; this is done by evaluating the *crowding-distance* parameter for each solution. For each function to be optimised, the crowding distance of each solution is increased by the difference between its immediate predecessor and successor by that function's value (specially, the endpoints are given a crowding distance of $+\infty$). The procedure is outlined in Algorithm 6; its time complexity is $O(m \cdot n \log n)$ (dominated by sorting). For a visualisation of the crowding distance in 2D space, refer to Figure 3.8.

Algorithm 6 Crowding distance assignment

Input: A nondominated front of solutions, $\mathcal{I}$.

procedure CROWDINGDISTANCEASSIGNMENT($\mathcal{I}$) ▷ Assign crowding distances to $\mathcal{I}$
  $l \leftarrow |\mathcal{I}|$ ▷ The number of solutions in the front
  for $i \leftarrow 1$ **to** l **do** $\mathcal{I}_i.d \leftarrow 0$ ▷ Initialise crowding distances
  end for
  for all f to optimise **do**
    SORT($\mathcal{I}$, f) ▷ Sort the elements of $\mathcal{I}$ by the values of f
    $\mathcal{I}_1.d \leftarrow \mathcal{I}_l.d \leftarrow +\infty$ ▷ Ensure that the endpoints are always preferred
    for $i \leftarrow 2$ **to** $l - 1$ **do** ▷ Increase the crowding distance for all other points
      $\mathcal{I}_i.d \leftarrow \mathcal{I}_i.d + (\mathcal{I}_{i+1}.f - \mathcal{I}_{i-1}.f)$
    end for
  end for
end procedure

Note that this, in combination with the previously defined $\succ$ relation, defines a new partial order, $\succ_n$ (the *crowded-comparison* relation), as follows:

$$p \succ_n q \iff (p \succ q) \vee (p \not\prec q \wedge p.d > q.d)$$

i.e. within a single nondominated front, the solutions that are *further* away from their immediate neighbours are preferred.

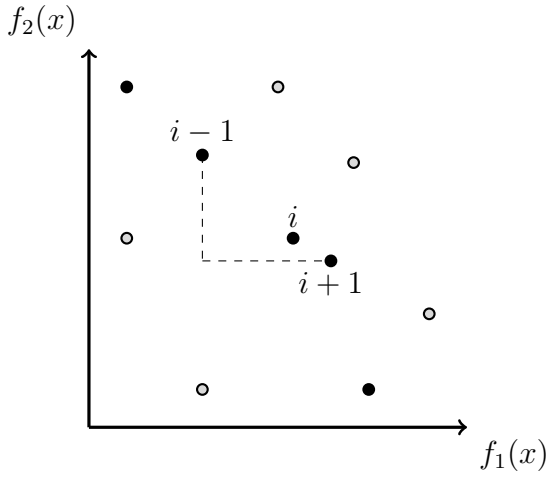


Figure 3.8: Example of crowding distance; for solution i it corresponds to the length of the dashed line.

Once P_{t+1} is produced, the next step is *combining* its elements to produce Q_{t+1}. This is done by a *randomised binary tournament* strategy:

1. Two pairs of indices (i, j) and (k, l) are picked uniformly at random, each index from the range $[1, n]$;
2. Solutions p and q are picked such that:
   (a) $p = \begin{cases} (P_{t+1})_i & (P_{t+1})_i \succ_n (P_{t+1})_j \\ (P_{t+1})_j & \text{otherwise} \end{cases}$
   (b) $q = \begin{cases} (P_{t+1})_k & (P_{t+1})_k \succ_n (P_{t+1})_l \\ (P_{t+1})_l & \text{otherwise} \end{cases}$
3. New solutions p' and q' are produced from p and q using the methods of *crossover* and *mutation*; these solutions are then added to Q_{t+1}.
4. The algorithm is repeated until n solutions have been produced.

There are many ways of performing crossover and mutation; NSGA-II utilises *simulated binary crossover (SBX)* and *polynomial mutation* [14]. As several parameters need to be introduced to properly define these operators, their respective formulae are omitted from this section; they may be found in Appendix A.3.

The combining algorithm is outlined in Algorithm 7, and, as this is the final building block, the full NSGA-II pseudocode is given in Algorithm 8. Its time complexity is dominated by the nondominated sorting subroutine, and as such it is $O(m \cdot n^2)$.

Algorithm 7 Solution combining algorithm

Input: A set of solutions, P.

function MAKENEWPOPULATION(P) ▷ Generate a set of solutions Q, from P
  for $x \leftarrow 1$ **to** $\frac{n}{2}$ **do**
    $i, j, k, l \leftarrow$ RANDOM($[1, n]$) ▷ Choose competitors uniformly at random
    if $P_i \succ_n P_j$ **then** $p \leftarrow P_i$ ▷ Choose the first parent solution
    else $p \leftarrow P_j$
    end if
    if $P_k \succ_n P_l$ **then** $q \leftarrow P_k$ ▷ Choose the second parent solution
    else $q \leftarrow P_l$
    end if
    $(p', q') \leftarrow$ CROSSOVER(p, q) ▷ Derive two child solutions via crossover
    MUTATE(p') ▷ Mutate the child solutions
    MUTATE(q')
    $Q \leftarrow Q \cup \{p', q'\}$ ▷ Add the child solutions to the set Q
  end for
  return Q
end function

Output: The set of new solutions, Q.

Algorithm 8 NSGA-II

Input: A sequence of functions, $\{f_i\}_{i=1}^{m}$, to optimise, population size n, number of iterations *iter*.

function NSGA-II($\vec{f}$, n, *iter*) ▷ Optimise the given set of functions
  $P_0 \leftarrow$ Generate(n) ▷ Generate n solutions uniformly at random
  $Q_0 \leftarrow$ MakeNewPopulation(P_0) ▷ Generate n child solutions
  for $t \leftarrow 0$ **to** $iter - 1$ **do**
    $R_t \leftarrow P_t \cup Q_t$ ▷ Combine parent and children solutions
    $\vec{\mathcal{F}} \leftarrow$ FastNondominatedSort(R_t) ▷ Find the nondominated fronts
    $P_{t+1} \leftarrow \emptyset$ ▷ Initialise the next parent solution set
    $i \leftarrow 1$
    while $|P_{t+1}| + |\mathcal{F}_i| \leq n$ **do** ▷ Front $\mathcal{F}_i$ fits completely in the next set
      CrowdingDistanceAssignment($\mathcal{F}_i$) ▷ Calculate crowding distances
      $P_{t+1} \leftarrow P_{t+1} \cup \mathcal{F}_i$ ▷ Add the front to the parent solution set
      $i \leftarrow i + 1$
    end while
    CrowdingDistanceAssignment($\mathcal{F}_i$) ▷ Calculate crowding distances
    Sort($\mathcal{F}_i$, $\succ_n$) ▷ Sort the final front by the crowded-comparison relation
    $P_{t+1} \leftarrow P_{t+1} \cup \mathcal{F}_i[1..(n - |P_{t+1}|)]$ ▷ Fill the remainder of the parent set
    $Q_{t+1} \leftarrow$ MakeNewPopulation(P_{t+1}) ▷ Generate n child solutions
  end for
  return $P_{iter} \cup Q_{iter}$
end function

Output: The solution set obtained after *iter* iterations, $P_{iter} \cup Q_{iter}$.

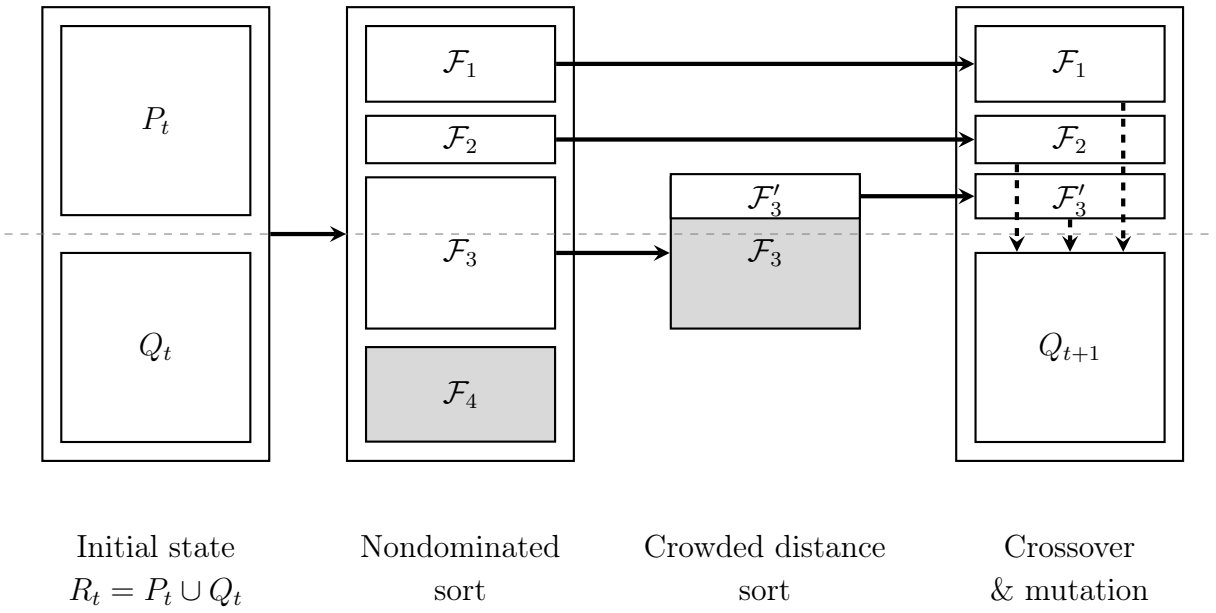


Figure 3.9: Visualisation of the stages within a single iteration of NSGA-II.

3.3.3.3 Training algorithm

All the components necessary for the full implementation of a training algorithm for the multiplex chain GMHMM are now in place. From everything defined thus far, a training procedure may be easily defined as such:

1. Initially, perform *separate training* on all of the chains in the multiplex, using one of the individual data types involved per chain;
2. Use NSGA-II to *extract* a set of (Pareto-optimal) solutions $\boldsymbol{\Omega}$ for the interlayer matrix $\boldsymbol{\omega}$, that optimise the individual log-likelihoods of each training sequence;
3. *Choose* one of the extracted solutions according to a (user-defined) metric.

I have chosen a metric that assigns larger significance to sequences with lower likelihoods, led by the idea that it is typically better to classify more sequences correctly (getting *closer to the general case*) than to have superb likelihood on fewer sequences (corresponding to *overfitting* the model).

The metric I used is given by the formula

$$fit(\boldsymbol{\omega}) \stackrel{\text{def}}{=} \sum_{i=1}^{k} i\mathbb{L}(s_i|\boldsymbol{\Theta}, \boldsymbol{\omega})$$

where the k training sequences have been sorted by their likelihoods in a *descending* order (hence the most likely sequence gets a factor of 1, while the least likely sequence gets a factor of k). The solution that is chosen as the final one is

$$\boldsymbol{\omega}^* = \underset{\boldsymbol{\omega}\in\boldsymbol{\Omega}}{\operatorname{argmax}}\, fit(\boldsymbol{\omega})$$

The full training algorithm is summarised in Algorithm 9.

Algorithm 9 Multiplex chain GMHMM training algorithm

Input: Multiplex chain GMHMM $M = (n, m, L, \vec{\boldsymbol{\Theta}}, \boldsymbol{\omega})$, training set $\vec{s}$ of size k, with L data types.

procedure TrainMuxChn(M, $\vec{s}$) ▷ Train the multiplex chain GMHMM using $\vec{s}$
  for $i \leftarrow 1$ **to** L **do**
    TrainChn($\boldsymbol{\Theta}_i$, $\vec{s^i}$, d, 10^{-6}) ▷ Train ith chain using the ith data type
  end for
  for $i \leftarrow 1$ **to** k **do** ▷ Extracting the functions to optimise
    $f_i(\boldsymbol{\omega}) \leftarrow \mathbb{L}(s_i|\boldsymbol{\Theta}, \boldsymbol{\omega})$ ▷ $\mathbb{L}(s_i|\boldsymbol{\Theta}, \boldsymbol{\omega})$ evaluated using the forward algorithm
  end for
  $\boldsymbol{\Omega} \leftarrow$ NSGA-II($\vec{f}$, *pop*, *iter*) ▷ Obtain Pareto-optimal solution set
  $\boldsymbol{\omega} \leftarrow \underset{\boldsymbol{\omega}'\in\boldsymbol{\Omega}}{\operatorname{argmax}} \sum_{i=1}^{k} i\mathbb{L}(s_i|\boldsymbol{\Theta}, \boldsymbol{\omega}')$ ▷ Choose one of the solutions according to fit
end procedure

A notable part of implementing this algorithm has been extracting the relevant objectives to optimise, as it utilises one of the major features introduced in C++11—*lambdas*.

I have implemented the NSGA-II algorithm to accept an argument of type `vector<function<double(vector<double>)> >` for the set of functions to optimise. Within the training routine, I have filled this vector with lambda expressions that set the respective $\boldsymbol{\omega}$ values, call the forward algorithm and report its log-likelihood output (*negated*, because I implemented minimisation rather than maximisation). The complete section of code is given in Listing 3.4.

This approach is very helpful as it gives a completely *generic* training procedure which does not require hard-coding the functions or having to execute the genetic algorithm separately from the main routine.

```
// Define the lambdas that calculate likelihoods for a given omega
objectives.resize(train_set.size());
for (uint t=0;t<train_set.size();t++)
{
  objectives[t] = [this, t, &train_set] (vector<double> X) -> double
  {
    double **temp_omega = new double*[L];
    for (int i=0;i<L;i++)
    {
      temp_omega[i] = new double[L];
      for (int j=0;j<L;j++)
      {
        temp_omega[i][j] = X[i*L + j];
      }
    }

    set_omega(temp_omega);

    for (int i=0;i<L;i++) delete[] temp_omega[i];
    delete[] temp_omega;

    return -log_likelihood(train_set[t]);
  };
}
```

Listing 3.4: Code for extracting the functions to optimise, as part of the training.

3.4 Full multiplex GMHMM

After fully implementing the core deliverable, a fitting next step is to *extend it* to take advantage of the full power and flexibility of the unconstrained GMHMM. As the models are highly similar, this section will focus on outlining the main implementation differences between them, and then discussing the major differences in detail.

3.4.1 Implementation differences

In this subsection, I will present the differences between the multiplex chain GMHMM and the full multiplex GMHMM in the form of bullet points, for clarity.

3.4.1.1 State

- There are still L GMHMM layers involved, each one trained on a unique data type from the training sequences.
- The structure of each layer is no longer constrained to be a chain; an arbitrary configuration using n states is allowed. As such, the state transition matrix, $\mathbf{T}$, must be specified.
- As it is no longer required for the GMHMM to always follow a specific path, the interlayer edges may now point from a node to itself; the edges themselves are still stored in the form of a matrix, $\boldsymbol{\omega}$. This means that the transition probabilities within the ith layer, $\mathbf{T}_i$, will effectively be scaled by $\boldsymbol{\omega}_{ii}$.

3.4.1.2 Algorithms

- Likelihoods are calculated as before; using the forward algorithm (§3.3.2.2).
- Parameter estimation of $\vec{\mu}$ and $\vec{\sigma}$ is still performed using the method described in §3.3.1.2. However, as the states no longer need to have any fixed meaning, the sub-output emission probabilities $\mathbf{O}'$ are no longer trained in this way.
- The parameters $\vec{\pi}$, $\mathbf{T}$ and $\mathbf{O}'$ are to be trained using a generic unsupervised learning algorithm for HMMs; in my project I have employed the most commonly used *Baum-Welch* algorithm for this purpose.
- Training of the interlayer transition matrix $\boldsymbol{\omega}$ is performed using the same call to NSGA-II as described in §3.3.3.3.

3.4.2 Baum-Welch algorithm

Most of the differences given above are minor and represent simple extensions; the only major difference is contained within the training procedure for a single layer GMHMM; the **Baum-Welch algorithm** [37]. Its aim is to produce, from a given HMM $\boldsymbol{\Theta}$ and output sequence $(\vec{y}, \vec{id})$, a new HMM $\boldsymbol{\Theta}'$ that is more likely to produce $(\vec{y}, \vec{id})$; i.e.

$$\mathbb{L}(\vec{y}, \vec{id}|\boldsymbol{\Theta}') \geq \mathbb{L}(\vec{y}, \vec{id}|\boldsymbol{\Theta})$$

This procedure could then be iterated a set number of times, until the desirable level of convergence is achieved.

3.4.2.1 EM iteration

The Baum-Welch algorithm is an HMM-specific implementation of the more generalised *Expectation-Maximisation (EM)* algorithm [16]. A single iteration operates as follows:

1. *(E step)* Evaluate the probabilities of being in state x at time t, $\gamma_x(t)$:

$$\gamma_x(t) \stackrel{\text{def}}{=} \mathbb{P}(X_t = x | \vec{y}, \vec{id}, \mathbf{\Theta})$$

   and the probabilities of transitioning from state i to state j at time t, $\xi_{ij}(t)$:

$$\xi_{ij}(t) \stackrel{\text{def}}{=} \mathbb{P}(X_t = i, X_{t+1} = j | \vec{y}, \vec{id}, \mathbf{\Theta})$$

2. *(M step)* Update the parameters of $\mathbf{\Theta}$ as follows:

   (a) The start-state probabilities, π'_x, simply correspond to the probabilities of being in state x at time 1:

$$\pi'_x = \gamma_x(1)$$

   (b) The transition probabilities, $\mathbf{T}'_{ij}$, may be calculated as the probabilities that the model will move to state j, after previously being located in state i:

$$\mathbf{T}'_{ij} = \frac{\sum_{t=1}^{T-1} \xi_{ij}(t)}{\sum_{t=1}^{T-1} \gamma_i(t)}$$

   (c) The sub-output emission probabilities, $\mathbf{O''}_{xy'}$, are evaluated as the probabilities that sub-output y' will be produced, given that the model is in state x:

$$\mathbf{O''}_{xy'} = \frac{\sum_{t=1}^{T} \mathbb{I}(id_t = y')\gamma_x(t)}{\sum_{t=1}^{T} \gamma_x(t)}$$

   where $\mathbb{I}$ is the indicator function (1 if $id_t = y$, 0 otherwise).

   obtaining the new model $\mathbf{\Theta}' = (\vec{\pi}', \mathbf{T}', \mathbf{O''})$.

In order to properly perform the *E step*, additional information is needed.

The forward algorithm provides us with $\alpha_t(x)$, the likelihoods of ending up in state x at time t and producing the first t elements of the output sequence. However, for $\gamma_x(t)$ it is needed to evaluate this probability conditional on the *entire* output sequence. Clearly, an algorithm is needed to evaluate the likelihoods of producing the corresponding *suffices* of the output sequence, assuming we start in state x. This algorithm will be described in the next paragraph.

3.4.2.2 Backward algorithm

The quantities referred to at the end of the previous part, the **backward likelihoods** $\beta_t(x)$, are the likelihoods of observing the output sequence from time $t+1$ onwards, assuming we are in state x at time t, i.e.

$$\beta_t(x) \stackrel{\text{def}}{=} \mathbb{L}(\{(y_i, id_i)\}_{i=t+1}^{T} | x_t = x, \mathbf{\Theta})$$

The backward likelihoods may efficiently be computed by the **backward algorithm**, which could be imagined as a *"reversed"* version of the forward algorithm.

The *base cases* are this time $\beta_T(x) = 1$, as the probability of observing an empty sequence is 1, regardless of which state we start from. The recurrence relation used to calculate $\beta_t(x)$, given that $\beta_{t+1}(x)$ had already been calculated for all x, may be derived (in a similar way as was done for $\alpha_t(x)$) to be:

$$\begin{aligned}
\beta_t(x) &= \mathbb{L}(\{(y_i, id_i)\}_{i=t+1}^{T} | x_t = x, \mathbf{\Theta}) \\
&= \sum_{x' \in S} \mathbb{L}(\{(y_i, id_i)\}_{i=t+1}^{T}, x_{t+1} = x' | x_t = x, \mathbf{\Theta}) \\
&= \sum_{x' \in S} \mathbb{L}(x_{t+1} = x' | x_t = x, \mathbf{\Theta}) \mathbb{L}(\{(y_i, id_i)\}_{i=t+1}^{T} | x_{t+1} = x', x_t = x, \mathbf{\Theta}) \\
&= \sum_{x' \in S} \mathbf{T}_{xx'} \mathbb{L}(\{(y_i, id_i)\}_{i=t+1}^{T} | x_{t+1} = x', \mathbf{\Theta}) \\
&= \sum_{x' \in S} \mathbf{T}_{xx'} \mathbb{L}((y_{t+1}, id_{t+1}) | x_{t+1} = x', \mathbf{\Theta}) \mathbb{L}(\{(y_i, id_i)\}_{i=t+2}^{T} | x_{t+1} = x', \mathbf{\Theta}) \\
&= \sum_{x' \in S} \mathbf{T}_{xx'} \mathbf{O'}_{x', id_{t+1}} \mathcal{N}(y_{t+1}; \mu_{id_{t+1}}, \sigma_{id_{t+1}}) \beta_{t+1}(x')
\end{aligned}$$

Similar underflow-avoiding techniques could be utilised for the backward algorithm as for the forward algorithm; namely, it is common to re-use the scaling coefficients $\vec{c}$ used to scale $\alpha_t(x)$, and let the actually stored value be

$$\hat{\beta}_t(x) = c_{t+1} \beta_t(x)$$

(specially, $\beta_T(x)$ is not scaled). Note that scaling by c_{t+1} does not need to normalise the values of $\beta_t(x)$, however it does simplify the E step performed afterwards.

The pseudocode of the entire procedure is highly similar to the forward algorithm (given in Algorithm 4), and as such is omitted from this paragraph; the time complexity of the backward algorithm is $O(n^2 \cdot T)$, same as for the forward algorithm.

3.4.2.3 E step

Now that the backward likelihoods have been evaluated, it is possible to combine them with the forward likelihoods to obtain the quantities $\gamma_x(t)$ and $\xi_{ij}(t)$. The probability $\gamma_x(t)$ may be evaluated as the *normalised likelihood* of being in state x at time t:

$$\gamma_x(t) = \mathbb{P}(X_t = x | \vec{y}, \vec{id}, \mathbf{\Theta}) = \frac{\mathbb{L}(X_t = x | \vec{y}, \vec{id}, \mathbf{\Theta})}{\sum_{x' \in s} \mathbb{L}(X_t = x' | \vec{y}, \vec{id}, \mathbf{\Theta})}$$

$\mathbb{L}(X_t = x|\vec{y}, \vec{id}, \mathbf{\Theta})$ may be evaluated using Bayes' theorem:

$$\begin{aligned}
\mathbb{L}(X_t = x|\vec{y}, \vec{id}, \mathbf{\Theta}) &= \frac{\mathbb{L}(\vec{y}, \vec{id}|X_t = x, \mathbf{\Theta})\mathbb{L}(X_t = x|\mathbf{\Theta})}{\mathbb{L}(\vec{y}, \vec{id}|\mathbf{\Theta})} \\
&= \frac{\mathbb{L}(\{(y_i, id_i)\}_{i=1}^{t}|X_t = x, \mathbf{\Theta})\mathbb{L}(X_t = x|\mathbf{\Theta})\mathbb{L}(\{(y_i, id_i)\}_{i=t+1}^{T}|X_t = x, \mathbf{\Theta})}{\mathbb{L}(\vec{y}, \vec{id}|\mathbf{\Theta})} \\
&= \frac{\mathbb{L}(\{(y_i, id_i)\}_{i=1}^{t}, X_t = x|\mathbf{\Theta})\mathbb{L}(\{(y_i, id_i)\}_{i=t+1}^{T}|X_t = x, \mathbf{\Theta})}{\mathbb{L}(\vec{y}, \vec{id}|\mathbf{\Theta})} \\
&= \frac{\alpha_t(x)\beta_t(x)}{\mathbb{L}(\vec{y}, \vec{id}|\mathbf{\Theta})}
\end{aligned}$$

Expressing $\mathbb{L}(\vec{y}, \vec{id}|\mathbf{\Theta})$ using the scaling coefficients $\vec{c}$:

$$\begin{aligned}
\mathbb{L}(X_t = x|\vec{y}, \vec{id}, \mathbf{\Theta}) &= \frac{\alpha_t(x)\beta_t(x)}{\mathbb{L}(\vec{y}, \vec{id}|\mathbf{\Theta})} \\
&= \frac{\alpha_t(x)\beta_t(x)}{\prod_{s=1}^{T} c_s^{-1}} \\
&= \left(\alpha_t(x)\prod_{s=1}^{t} c_s\right)\left(\beta_t(x)\prod_{s=t+1}^{T} c_s\right) = \hat{\alpha}_t(x)\hat{\beta}_t(x)
\end{aligned}$$

Therefore, $\gamma_x(t)$ may be simply calculated as

$$\gamma_x(t) = \frac{\hat{\alpha}_t(x)\hat{\beta}_t(x)}{\sum_{x' \in S} \hat{\alpha}_t(x')\hat{\beta}_t(x')}$$

Similarly, the probability $\xi_{ij}(t)$ may be interpreted as the normalised likelihood of taking the transition $i \longrightarrow j$ at time t:

$$\xi_{ij}(t) = \mathbb{P}(X_t = i, X_{t+1} = j|\vec{y}, \vec{id}, \mathbf{\Theta}) = \frac{\mathbb{L}(X_t = i, X_{t+1} = j|\vec{y}, \vec{id}, \mathbf{\Theta})}{\sum_{a \in S}\sum_{b \in S} \mathbb{L}(X_t = a, X_{t+1} = b|\vec{y}, \vec{id}, \mathbf{\Theta})}$$

Using the model assumptions and Bayes' theorem, $\mathbb{L}(X_t = i, X_{t+1} = j|\vec{y}, \vec{id}, \mathbf{\Theta})$ may also be defined in terms of the parameters we already know (steps omitted—refer to Figure 3.10 for an illustration of the formula):

$$\begin{aligned}
\mathbb{L}(X_t = i, X_{t+1} = j|\vec{y}, \vec{id}, \mathbf{\Theta}) &= \frac{\alpha_t(i)\mathbf{T}_{ij}\mathbf{O'}_{j,id_{t+1}}\mathcal{N}(y_{t+1}; \mu_{id_{t+1}}, \sigma_{id_{t+1}})\beta_{t+1}(j)}{\prod_{s=1}^{T} c_s^{-1}} \\
&= c_{t+1}\hat{\alpha}_t(i)\mathbf{T}_{ij}\mathbf{O'}_{j,id_{t+1}}\mathcal{N}(y_{t+1}; \mu_{id_{t+1}}, \sigma_{id_{t+1}})\hat{\beta}_{t+1}(j)
\end{aligned}$$

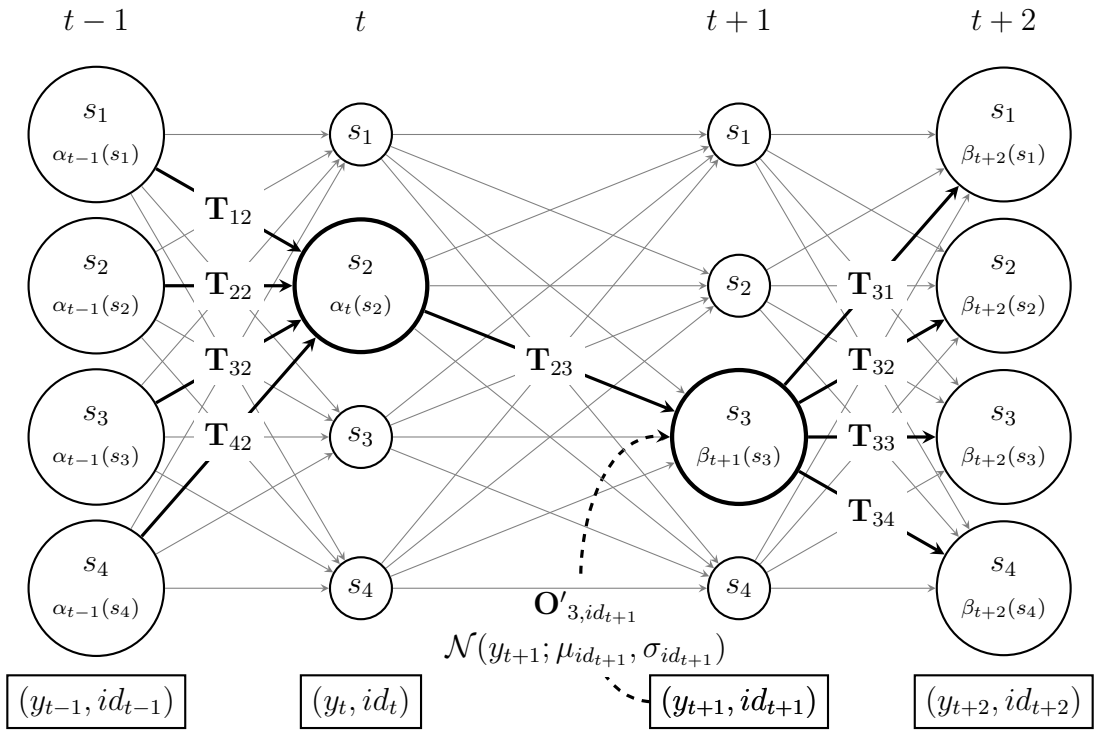


Figure 3.10: Illustration of the formula used to calculate $\mathbb{L}(X_t = i, X_{t+1} = j|\vec{y}, \vec{id}, \mathbf{\Theta})$, on a four-state GMHMM, with $i = s_2$ and $j = s_3$.

Therefore, $\xi_{ij}(t)$ may be calculated as

$$\begin{aligned}
\xi_{ij}(t) &= \frac{c_{t+1}\hat{\alpha}_t(i)\mathbf{T}_{ij}\mathbf{O'}_{j,id_{t+1}}\mathcal{N}(y_{t+1};\mu_{id_{t+1}},\sigma_{id_{t+1}})\hat{\beta}_{t+1}(j)}{\sum_{a\in S}\sum_{b\in S} c_{t+1}\hat{\alpha}_t(a)\mathbf{T}_{ab}\mathbf{O'}_{b,id_{t+1}}\mathcal{N}(y_{t+1};\mu_{id_{t+1}},\sigma_{id_{t+1}})\hat{\beta}_{t+1}(b)} \\
&= \frac{c_{t+1}\hat{\alpha}_t(i)\mathbf{T}_{ij}\mathbf{O'}_{j,id_{t+1}}\mathcal{N}(y_{t+1};\mu_{id_{t+1}},\sigma_{id_{t+1}})\hat{\beta}_{t+1}(j)}{\sum_{a\in S}\hat{\alpha}_t(a)c_{t+1}\sum_{b\in S}\mathbf{T}_{ab}\mathbf{O'}_{b,id_{t+1}}\mathcal{N}(y_{t+1};\mu_{id_{t+1}},\sigma_{id_{t+1}})\hat{\beta}_{t+1}(b)} \\
&= \frac{c_{t+1}\hat{\alpha}_t(i)\mathbf{T}_{ij}\mathbf{O'}_{j,id_{t+1}}\mathcal{N}(y_{t+1};\mu_{id_{t+1}},\sigma_{id_{t+1}})\hat{\beta}_{t+1}(j)}{\sum_{a\in S}\hat{\alpha}_t(a)\hat{\beta}_t(a)}
\end{aligned}$$

This concludes the E step of the algorithm.

3.4.2.4 Full algorithm

With a method of computing $\gamma_x(t)$ and $\xi_{ij}(t)$ in place, the M step is trivially performed by just evaluating the formulae given in §3.4.2.1. With this in mind, the pseudocode of the Baum-Welch algorithm is given in Algorithm 10.

The time complexity of the forward-backward step is $O(n^2 \cdot T)$, as previously discussed; the E step requires $O(n^2 \cdot T)$ time, and the M step requires $O((n^2 + n \cdot m) \cdot T)$ time. Therefore, the time complexity of a single Baum-Welch iteration is dominated by the M step, i.e. $O((n^2 + n \cdot m) \cdot T)$. Storing the function $\xi_{ij}(t)$ would require a space

complexity of $O(n^2 \cdot T)$, however, storing this parameter is not necessary (as it may be computed ad-hoc during the M step); hence the iteration's overall space complexity equals that of the forward/backward algorithms.

Algorithm 10 Baum-Welch algorithm

Input: GMHMM $\mathbf{\Theta} = (n, m, \vec{\pi}, \mathbf{T}, \mathbf{O}', \vec{\mu}, \vec{\sigma})$, output sequence $(\vec{y}, \vec{id})$ of length T.

procedure BAUMWELCH($\mathbf{\Theta}$, $\vec{y}$, $\vec{id}$, T) ▷ Generate a more fit model wrt $(\vec{y}, \vec{id})$
  $(\hat{\alpha}, \vec{c}, \ell) \leftarrow$ FORWARD($\mathbf{\Theta}$, $\vec{y}$, $\vec{id}$, T) ▷ Run the forward algorithm
  $\hat{\beta} \leftarrow$ BACKWARD($\mathbf{\Theta}$, $\vec{y}$, $\vec{id}$, T, $\vec{c}$) ▷ Run the backward algorithm, reusing $\vec{c}$
  for $t \leftarrow 1$ **to** T **do** ▷ E step; calculating $\gamma_x(t)$ and $\xi_{ij}(t)$
    for $i \leftarrow 1$ **to** n **do**
      $s \leftarrow \sum_{x \in S} \hat{\alpha}_t(x)\hat{\beta}_t(x)$
      $\gamma_i(t) \leftarrow \frac{\hat{\alpha}_t(i)\hat{\beta}_t(i)}{s}$
      for $j \leftarrow 1$ **to** n **do**
        $\xi_{ij}(t) \leftarrow \frac{c_{t+1}\hat{\alpha}_t(i)\mathbf{T}_{ij}\mathbf{O'}_{j,id_{t+1}}\mathcal{N}(y_{t+1};\mu_{id_{t+1}},\sigma_{id_{t+1}})\hat{\beta}_{t+1}(j)}{s}$
      end for
    end for
  end for
  for $i \leftarrow 1$ **to** n **do** ▷ M step; reestimating $\vec{\pi}$, $\mathbf{T}$ and $\mathbf{O}'$
    $\pi_i \leftarrow \gamma_1(i)$
    for $j \leftarrow 1$ **to** n **do**
      $\mathbf{T}_{ij} \leftarrow \frac{\sum_{t=1}^{T-1} \xi_{ij}(t)}{\sum_{t=1}^{T-1} \gamma_i(t)}$
    end for
    for $y' \leftarrow 1$ **to** m **do**
      $\mathbf{O'}_{iy'} \leftarrow \frac{\sum_{t=1}^{T} \mathbb{I}(id_t = y')\gamma_i(t)}{\sum_{t=1}^{T} \gamma_i(t)}$
    end for
  end for
end procedure

3.4.2.5 Multiple sequence training

Thus far, I have discussed an implementation of the Baum-Welch algorithm that takes only a **single** sequence for training. However, as previously discussed, within the context of my project it is more likely that *multiple shorter sequences* will be used. I will hence conclude the discussion of the Baum-Welch algorithm with an overview of extending it for training on k separate sequences [37].

Luckily, as the re-estimation formulae for $\mathbf{T}_{ij}$ and $\mathbf{O'}_{iy'}$ simply measure the frequencies of their respective transitions and emissions, it suffices to run the forward-backward procedure and E step on each of the sequences (producing $\gamma_x^\ell(t)$ and $\xi_{ij}^\ell(t)$ for the ℓ-th

sequence), and then simply summing up contributions from every sequence:

$$\mathbf{T}'_{ij} = \frac{\sum_{\ell=1}^{k}\sum_{t=1}^{T-1}\xi_{ij}^{\ell}(t)}{\sum_{\ell=1}^{k}\sum_{t=1}^{T-1}\gamma_{i}^{\ell}(t)}$$

$$\mathbf{O}''_{xy'} = \frac{\sum_{\ell=1}^{k}\sum_{t=1}^{T}\mathbb{I}(id_{t}^{\ell}=y')\gamma_{x}^{\ell}(t)}{\sum_{\ell=1}^{k}\sum_{t=1}^{T}\gamma_{x}^{\ell}(t)}$$

The formula for re-estimating π_x may also be trivially modified, by considering an equiprobable sum of each of the sequences' initial state distributions:

$$\pi'_{x} = \frac{1}{k}\sum_{\ell=1}^{k}\gamma_{x}^{\ell}(1)$$

It should be clear that very few changes are needed to Algorithm 10 in order to make a BaumWelch procedure capable of handling multiple sequences; as such, the full pseudocode is omitted from this paragraph.

3.4.3 Training

Having implemented the Baum-Welch algorithm, I had all the necessary pieces to finalise the training algorithm; it is summarised in Algorithm 11.

Algorithm 11 Full multiplex GMHMM training algorithm

Input: Full multiplex GMHMM $M = (n, m, L, \vec{\mathbf{\Theta}}, \boldsymbol{\omega})$, training set $\vec{s}$ of size k, and lengths $\vec{T}$ with L data types.

procedure TrainMuxGMHMM(M, $\vec{s}$, $\vec{T}$) ▷ Train the multiplex GMHMM using $\vec{s}$
  for $i \leftarrow 1$ **to** L **do**
    TrainGauss($\mathbf{\Theta}_i$, $\vec{s^i}$) ▷ Estimate the parameters $\vec{\mu}_i$ and $\vec{\sigma}_i$
    BaumWelch($\mathbf{\Theta}_i$, $\vec{s^i}$, $\vec{T^i}$) ▷ Train ith GMHMM using the ith data type
  end for
  for $i \leftarrow 1$ **to** k **do** ▷ Extracting the functions to optimise
    $f_i(\boldsymbol{\omega}) \leftarrow \mathbb{L}(s_i|\mathbf{\Theta}, \boldsymbol{\omega})$ ▷ $\mathbb{L}(s_i|\mathbf{\Theta}, \boldsymbol{\omega})$ evaluated using the forward algorithm
  end for
  $\mathbf{\Omega} \leftarrow$ NSGA-II($\vec{f}$, pop, $iter$) ▷ Obtain Pareto-optimal solution set
  $\boldsymbol{\omega} \leftarrow \underset{\boldsymbol{\omega}' \in \mathbf{\Omega}}{\operatorname{argmax}} \sum_{i=1}^{k} i\mathbb{L}(s_i|\mathbf{\Theta}, \boldsymbol{\omega}')$ ▷ Choose one of the solutions according to fit
end procedure

Note that the algorithm as presented shares many similarities with the chain GMHMM training algorithm (§ 3.3.3.3); the only difference is the way the individual layers are trained. With this I conclude the discussion of the full multiplex GMHMM, and with it, the entire implementation phase of the project.

3.5 Summary

This chapter has summarised the entire implementation phase that resulted in successfully achieving the core and extension machine learning model deliverables; in particular, I have given a detailed discussion of the key algorithms used for evaluating output likelihoods and model training, and demonstrated how those building blocks may be combined into a model used for binary classification.

The next chapter will focus on the evaluation phase of the project, reviewing the methods used to ascertain the correct behaviour of the models, as well as examining the extent to which the implemented models have accomplished their success criteria.

Chapter 4

Evaluation

Having thoroughly reviewed the implementation of my project, in this chapter I will aim to demonstrate evidence that the project has successfully achieved all of the success criteria outlined in its original proposal.

This will include an overview of the methods I have used to verify that the implemented models behave as proposed, followed by a comparative evaluation using synthetic and biomolecular data (where I will demonstrate that introducing multiplexity produces a significant rise in classifier performance, as predicted), and assessment of the robustness of the models in presence of experimental error.

4.1 Success criteria

The evaluation of my project first required thinking back to the project's main success criteria outlined in its proposal (Appendix D), in order to be able to assess whether they have been achieved. They are summarised in this section, along with pointers to the areas of the dissertation where their successful achievement is demonstrated. I have denoted a successfully met criterion with a tick (✓), adding an additional tick if the criterion had been extended in some way.

Criterion 1: *The complete proposed classifier data structure should be implemented, incorporating at least the three main building blocks... [Chain GMHMM, Multiplex, NSGA-II].*
This refers to the successful core project implementation, which has been thoroughly demonstrated in §3.3. ✓ Furthermore, a completely generalised multiplex GMHMM was successfully implemented, as described in §3.4. ✓

Criterion 2: *Correct operation of the individual modules within the structure should be tested on the sample tests provided in relevant literature or academic papers.*
I have met this criterion by implementing a variety of *unit tests* on the key algorithms used in the models; these will be outlined in §4.2 and have been setup for both the core and extension components. ✓✓

Criterion 3: *The accuracy of the classifier should be evaluated with at least the two methods ... (supervised learning setup to estimate accuracy, and comparison with single-*

layered classifiers). It might prove useful to provide further ways of evaluation... This refers to successfully implementing an evaluation suite as prescribed by the criterion; I was successful in this as well, and the results of the comparative evaluation are given in §4.3. ✓ Furthermore, I have extended the suite with *noise testing*, in order to investigate the robustness of the models in the presence of experimental errors; the results of this analysis are summarised in §4.4. ✓

4.2 Unit tests

As prescribed by Criterion 2, I have utilised a variety of known examples from relevant academic papers, presentations or literature in order to properly verify that the principal algorithms are operating as expected. The key algorithms covered by unit testing, along with their respective source used for the test, are given in Table 4.1. The exact tests (with expected outputs) are presented in Appendix C.

Algorithm under test	**Test source**
Forward/backward algorithm	Russell & Norvig [38]
Baum-Welch algorithm	Frazzoli [19]
NSGA-II	Deb *et al.* [15]

Table 4.1: A summary of the unit tests on key algorithms within the project.

The unit tests have been implemented either as standalone C++ programs or shell scripts; a screenshot of recompiling and executing one of them is given in Figure 4.1.

4.3 Comparative evaluation

4.3.1 Performance metrics

As previously mentioned in §2.3, there exists a wide variety of established metrics for evaluating binary classifiers [36]. Calculating any one of them for a single classifier execution on a data set requires determining four quantities: the number of *true positives* (P^+), *true negatives* (N^+), *false positives* (P^-) and *false negatives* (N^-).

The metrics I have utilised for the purposes of evaluating my project are:

- **Accuracy**: the proportion of samples classified correctly:

$$accuracy = \frac{P^+ + N^+}{P^+ + N^+ + P^- + N^-}$$

- **Sensitivity**: the proportion of positive samples classified correctly:

$$sensitivity = \frac{P^+}{P^+ + N^-}$$

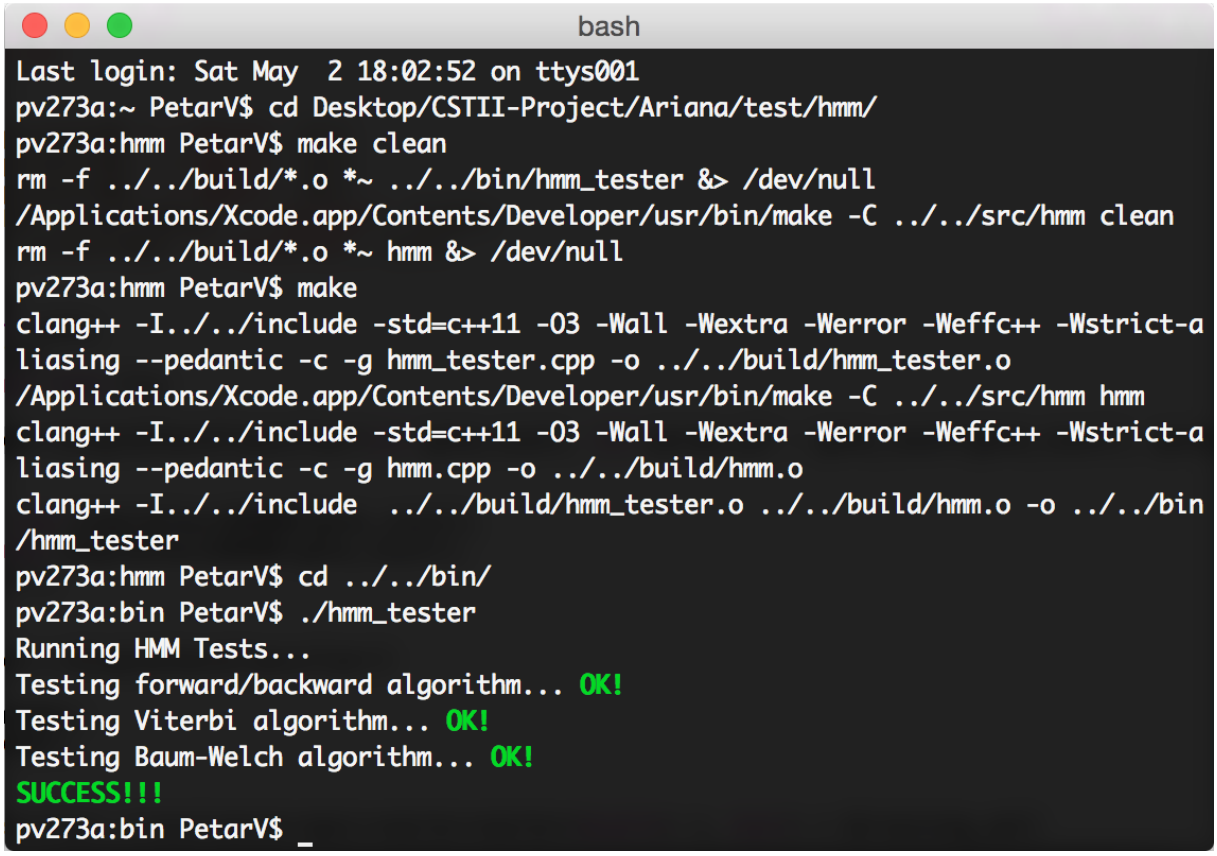


Figure 4.1: Workflow of recompiling and executing the HMM unit tests (also including tests on the unused Viterbi algorithm).

- **Matthews Correlation Coefficient (MCC)** [28]: the correlation coefficient between the observed and predicted classifications:

$$MCC = \frac{P^+N^+ - P^-N^-}{\sqrt{(P^+ + P^-)(P^+ + N^-)(N^+ + P^-)(N^+ + N^-)}}$$

- **$\mathbf{F_1}$ score**: the harmonic mean of precision (the proportion of positively classified samples that are positive; $precision = \frac{P^+}{P^+ + P^-}$) and sensitivity:

$$F_1 = \frac{2 \cdot precision \cdot sensitivity}{precision + sensitivity}$$

- **Receiver operating characteristic (ROC) curve**: a plot of the true positive rate (sensitivity) against the false positive rate ($FPR = \frac{P^-}{P^- + N^+}$) as the *classification threshold* (the difference in likelihood needed to classify as positive) is varied.

Accuracy, sensitivity, F_1 score, and the area under the ROC curve vary between 0 and 1; the MCC varies between -1 and 1 (higher values being indicative of a better classifier).

4.3.2 Experimental setup

Evaluating a classifier requires partitioning a set of labeled samples into a *training* and *testing* set, training it using the training set, and then calculating the aforementioned

metrics upon executing the trained classifier on the testing set.

A single such execution does not provide enough information to reliably compare two learning algorithms; to alleviate this, I have utilised **k-fold crossvalidation**—partitioning the data set into k equally-sized subsets and then averaging the obtained metrics over k runs; in each run, one of the subsets is used for testing while the remaining $k - 1$ are used for training (Figure 4.2). Furthermore, I have made the crossvalidation **stratified**; each of the k subsets has an approximately equal proportion of classes to the overall set (so that, for a particular class, all of the k runs will use roughly the same amount of data for training).

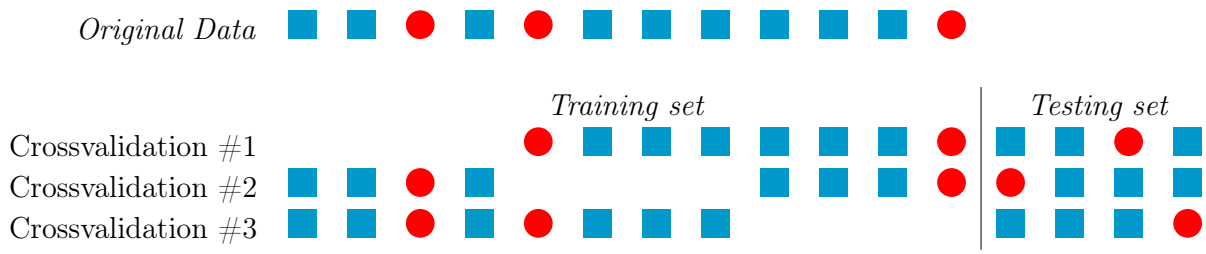


Figure 4.2: Illustration of a stratified three-fold crossvalidation.

When comparing two classifiers using crossvalidation, I have executed them on the *same* partitions into the training and test sets, allowing for inherently *pairing* the results obtained when executing each classifier on a particular partition. This means that a **paired t-test** may be utilised to assert *statistical significance* of the observed differences in classifier performance.

The paired t-test produces a p-value, representing the probability of a performance difference *greater than or equal to* the one observed, assuming that the *null hypothesis* (which assumes that the two classifiers are equal in expected value of the observed metric) is true. The null hypothesis is *rejected* (i.e. the observed differences between the classifiers are deemed *statistically significant*) when this probability is smaller than a given threshold; typically, $p < 0.01$ is chosen as the criterion for rejection.

4.3.3 Synthetic data

As it is generally a very hard problem to extract the *exact* correlations present within a real-life data set, I have decided to perform the first round of comparative evaluation on data that is purely synthetic. This gave me maximal control with respect to tweaking the parameters of the correlation between the data types and provided an additional sanity-check that the overall models behaved as expected.

The data set I will be discussing within this subsection consists of 2000 balanced samples (1000 per class), consisting of two types of data in each data point. There are $k = 5$ sub-outputs involved in producing the samples, by using a Gaussian distribution with standard deviation $\sigma = 1$. In the samples labeled with C_1, exactly one of the

sub-output means in each of the data types has been offset by σ compared to the C_2 samples. It is hence the job of the trained classifier to identify this anomalous offset—and the multiplex models will have access to the anomaly in both layers; as such, they are expected to perform significantly better than the monoplex network models on this data set.

The results of the comparative evaluation after 10-fold crossvalidation on this data set are summarised in Tables 4.2 and 4.3 as well as in Figure 4.3.

Parameter	**Chain GMHMM**	**Multiplex chain GMHMM**	p**-value**
Accuracy	0.8135	**0.8635**	0.0011
Sensitivity	0.831	**0.877**	0.0006
MCC	0.627733	**0.728227**	0.0010
F_1 score	0.816916	**0.865022**	0.0008

Table 4.2: Comparative evaluation results of the chain GMHMMs on synthetic data.

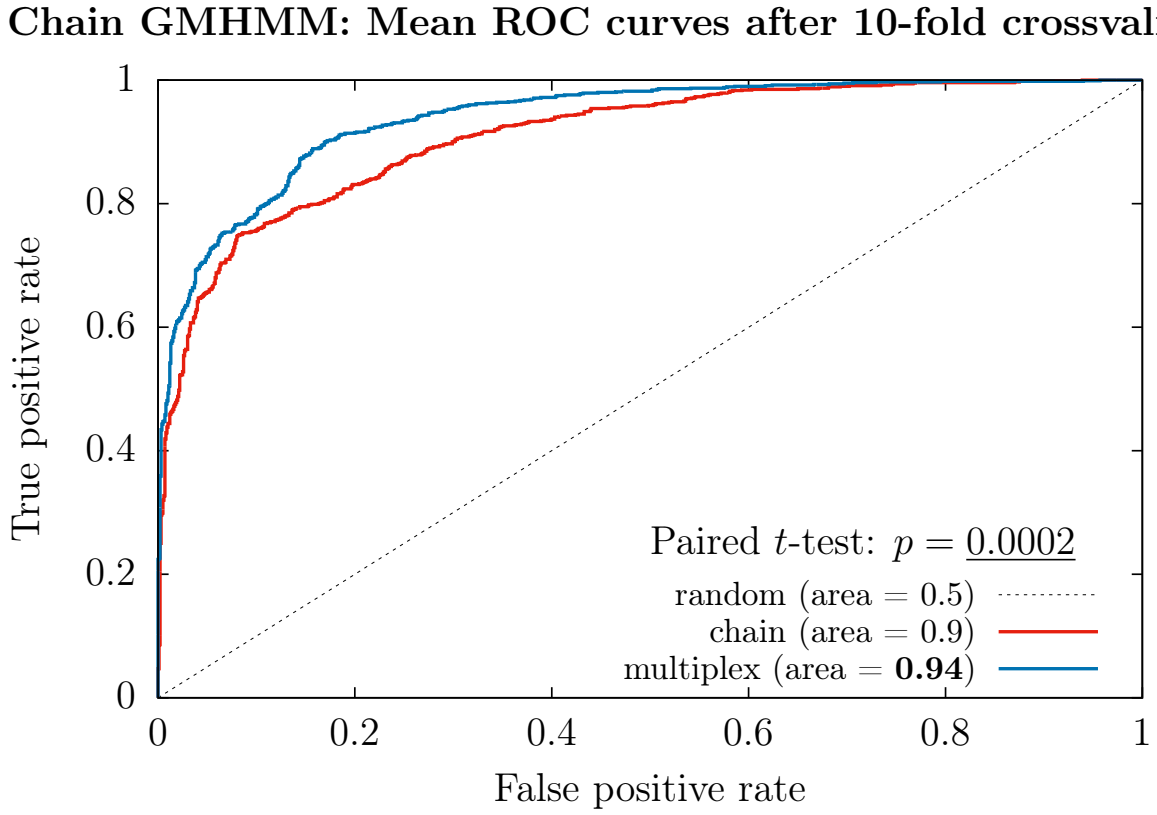
Parameter	**Full GMHMM**	**Full multiplex GMHMM**	p**-value**
Accuracy	0.842	**0.8605**	0.0090
Sensitivity	0.858	**0.877**	0.1250
MCC	0.685504	**0.724298**	0.0072
F_1 score	0.844418	**0.862484**	0.0048

Table 4.3: Comparative evaluation results of the full GMHMM models on synthetic data.

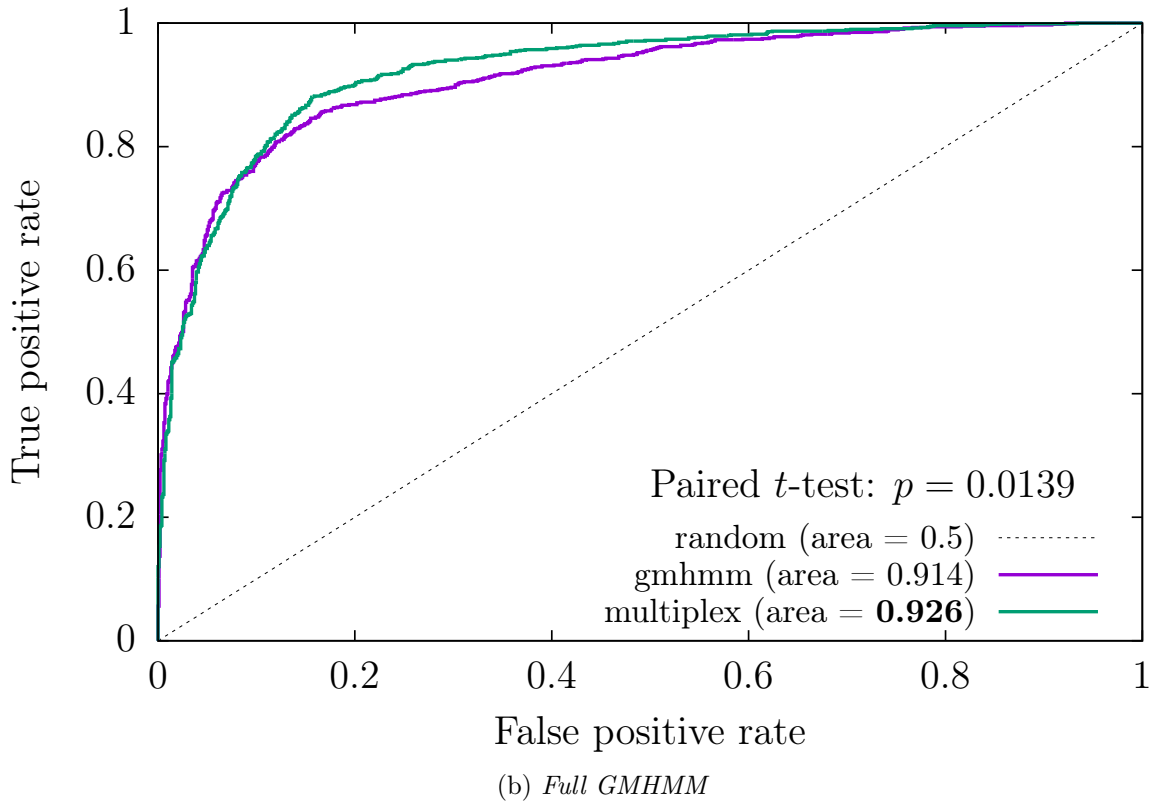
These results successfully demonstrate that, as expected, **the multiplex models outperform their monoplex counterparts** on each of the metrics considered—and all but two of those performance gains are statistically significant ($p < 0.01$). The decrease in significance for the full GMHMM may be explained by the fact that, as the data is normally distributed, there is bound to be a set of data points that will be offset incorrectly, and hence end up being classified to the other class, no matter how well the classifier is trained. The full GMHMM already manages to approach those limits on its own, and thus the multiplex GMHMM is not able to give as significant comparative benefits as the chain multiplex (however the improvements are still noticeable).

Lastly, it might be useful to compare the two monoplex and multiplex models against one another. The results of this comparison are summarised in Table 4.4.

The full GMHMM outperforms the chain GMHMM with statistical significance in almost all metrics, which is expected due to the larger expressivity of the full GMHMM. The observed differences in their multiplex counterparts, however, have been shown to be largely statistically insignificant, which is consistent with both models approaching the previously mentioned limits imposed by the data's underlying distribution.



(a) *Chain GMHMM*



(b) *Full GMHMM*

Figure 4.3: Mean ROC curves for the implemented models on synthetic data.

Parameter	Monoplex models	Multiplex models
Accuracy	$p = \underline{0.0013}$	$p = 0.2339$
Sensitivity	$p = 0.0166$	$p = 0.5000$
MCC	$p = \underline{0.0011}$	$p = 0.3192$
F_1 score	$p = \underline{0.0012}$	$p = 0.3059$
Area under ROC curve	$p = \underline{0.0073}$	$p = \underline{0.0021}$

Table 4.4: Statistical significance of the observed differences in the monoplex and multiplex models separately, on synthetic data.

4.3.4 Biomolecular data

Satisfied that the models operate as expected on the controlled data set, I moved on to evaluating the same performance metrics as before on a real data set.

The main problem tackled here is classification of subjects for *breast invasive carcinoma (BRCA)*, based on the activity metrics for the genes in the tumour necrosis factor receptor superfamily (TNFRSF). These genes are responsible for production of receptors that are able to bind to tumour necrosis factors (TNFs), proteins capable of inducing cell death (apoptosis) of tumourous cells, and as such, their activity levels are expected to be correlated with the incidence of specific cancers. The particular genetic activity metrics considered as the two principal data types are *gene expression* and *methylation*.

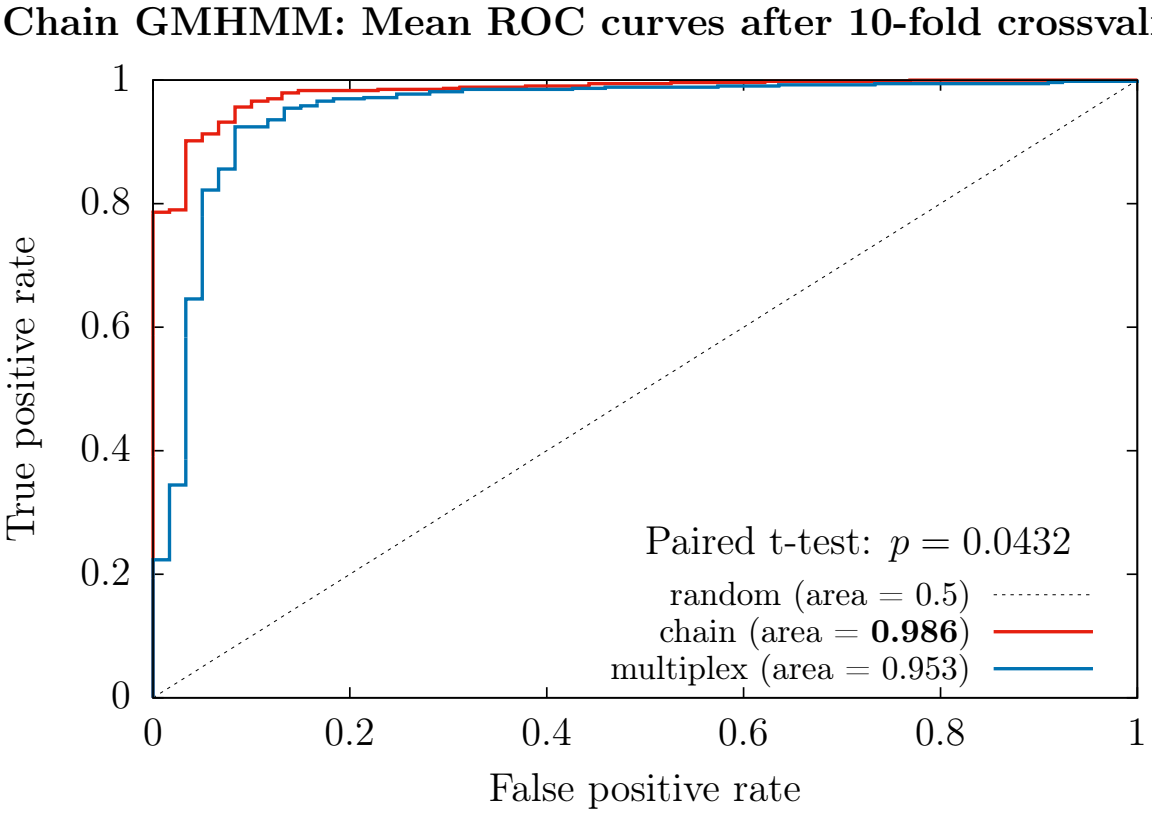
The data set utilised has been obtained from the *Cancer Genome Atlas (TCGA)*[1] and contains expression and methylation measurements for 26 genes in the TNFRSF; there are 776 patient and 79 normal samples in the data set. To translate this data into the GMHMM terminology, each gene corresponds to a single sub-output, and a single subject's sequence is fed into the models such that the more active genes come first (here I have used the *euclidean norm* of a gene's data point as its activity measure; $activity(g) = \sqrt{expression_g^2 + methylation_g^2}$).

The results of the comparative evaluation after 10-fold crossvalidation on this data set are summarised in Tables 4.5 and 4.6 as well as in Figure 4.4.

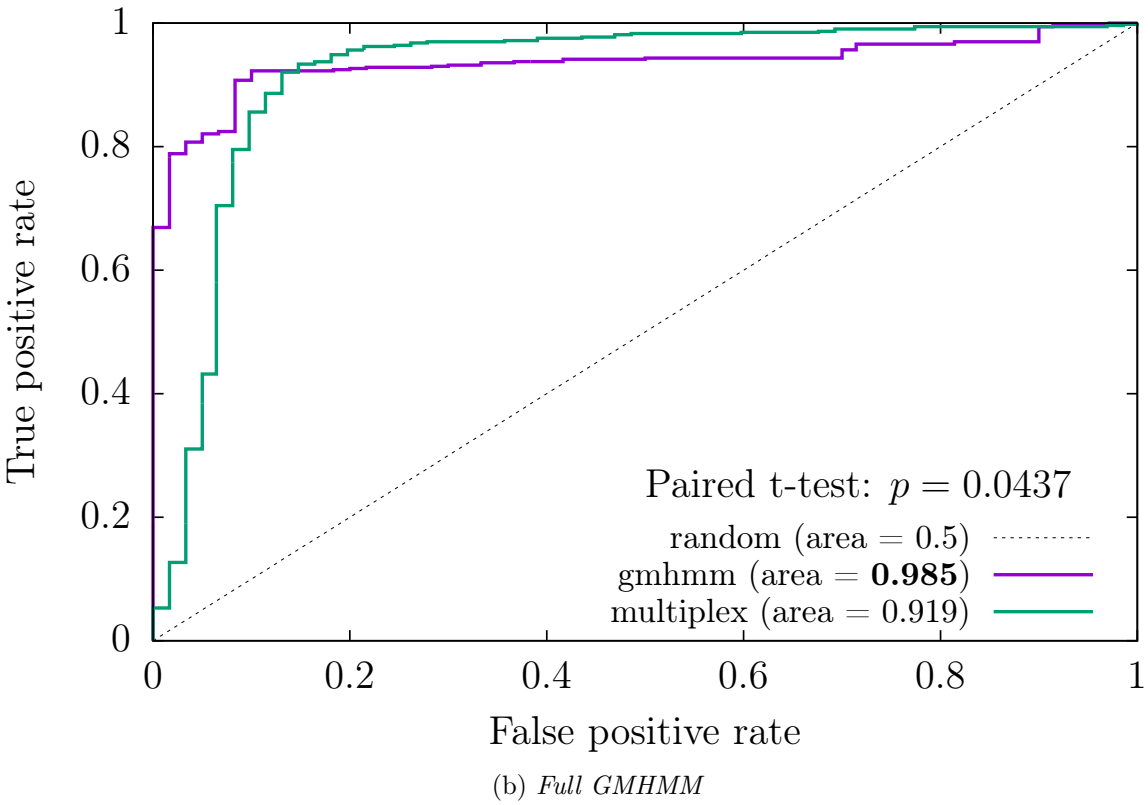
Parameter	Chain GMHMM	Multiplex chain GMHMM	p-value
Accuracy	0.898296	**0.944091**	$\underline{0.000063}$
Sensitivity	0.890239	**0.984906**	$\underline{0.000003}$
MCC	**0.665473**	0.663314	0.483599
F_1 score	0.939568	**0.969341**	$\underline{0.000039}$

Table 4.5: Comparative evaluation results of the chain GMHMMs on biomolecular data.

[1] https://tcga-data.nci.nih.gov/tcga/



(a) *Chain GMHMM*



(b) *Full GMHMM*

Figure 4.4: Mean ROC curves for the implemented models on biomolecular data.

Parameter	Full GMHMM	Full multiplex GMHMM	p-value
Accuracy	0.935643	**0.9424548**	0.3299
Sensitivity	0.941292	**0.9584906**	0.2474
MCC	0.734150	**0.7500727**	0.3130
F_1 score	0.962836	**0.9669872**	0.3329

Table 4.6: Comparative evaluation results of the full GMHMM models on biomolecular data.

These results once again demonstrate that, as expected, **the multiplex models outperform their monoplex counterparts** on most of the metrics considered. The exceptions are the MCC in the chain GMHMM case, as well as the area under the ROC curves in both cases. The decrease in the ROC curve integral is consistent with the formula used to discriminate the solutions returned by NSGA-II (given in §3.3.3.3), as it gives preference to having more sequences with sufficient likelihoods, rather than fewer sequences with very high likelihoods.

All the positive differences observed for the chain GMHMM models are statistically *highly significant*; the differences observed for the full GMHMM, while evident, are not statistically significant. This could once again be explained by the fact that the monoplex GMHMM classifier is already able to achieve very high performance on the given data (in several testing sets, *perfect classification* was observed), and therefore it is very hard to make a significant improvement. While outside of the scope of the goals of this project, this likely also demonstrates that the models I have implemented may be **highly suitable** for handling diagnostics problems such as this one.

The statistical significances of the differences observed between each the monoplex and multiplex models separately are given in Table 4.7. The significances obtained

Parameter	Monoplex models	Multiplex models
Accuracy	$p = \underline{0.0081}$	$p = 0.4606$
Sensitivity	$p = \underline{0.0009}$	$p = 0.1129$
MCC	$p = 0.0589$	$p = 0.3192$
F_1 score	$p = \underline{0.0070}$	$p = 0.4049$
Area under ROC curve	$p = 0.4534$	$p = 0.2092$

Table 4.7: Statistical significance of the observed differences in the monoplex and multiplex models separately, on biomolecular data.

from Table 4.7 further emphasise the *relative power* of the implemented models; the chain GMHMM is already a powerful model for this problem, and it is significantly outperformed by all of the other three models, that cannot be distinguished from one another with statistical significance—implying that it is possible that all of them have

reached the potential "limits" imposed by the data in the data set.

4.4 Robustness analysis

When dealing with real data sets, all the observations presented to the learning algorithms are necessarily accompanied with a certain experimental error, and it is crucial that the learning algorithm is, to a certain extent, capable of *discriminating the noise* from the actual data to be learned.

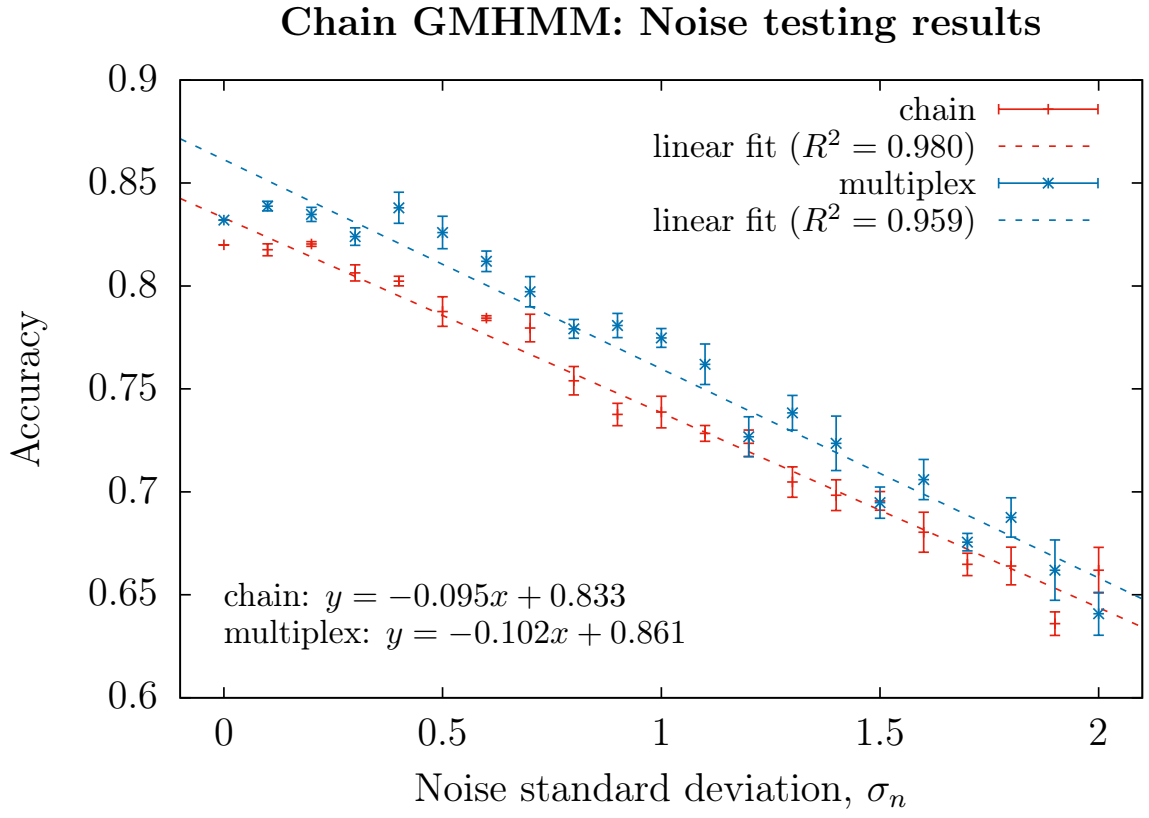
The findings from § 4.3.4 indicate that the models seem to perform rather well in the presence of any experimental errors that were present in the biomolecular data. To strengthen this indication, I have extended my evaluation suite with a simple *noise testing* facility, which is capable of introducing Gaussian noise (with a specified mean μ_n and standard deviation σ_n) into a data set before starting the crossvalidation run.

To be able to assess the robustness of the implemented models, I have used a synthetically generated data set with similar characteristics as the one described in §4.3.3, and recorded the accuracy of the models as the noise's standard deviation increases from 0 to 2, in increments of 0.1 (the mean of the noise was held at zero throughout the experiment). For each noise distribution, the sample mean of *ten 10-fold crossvalidation runs* was used as the final result.

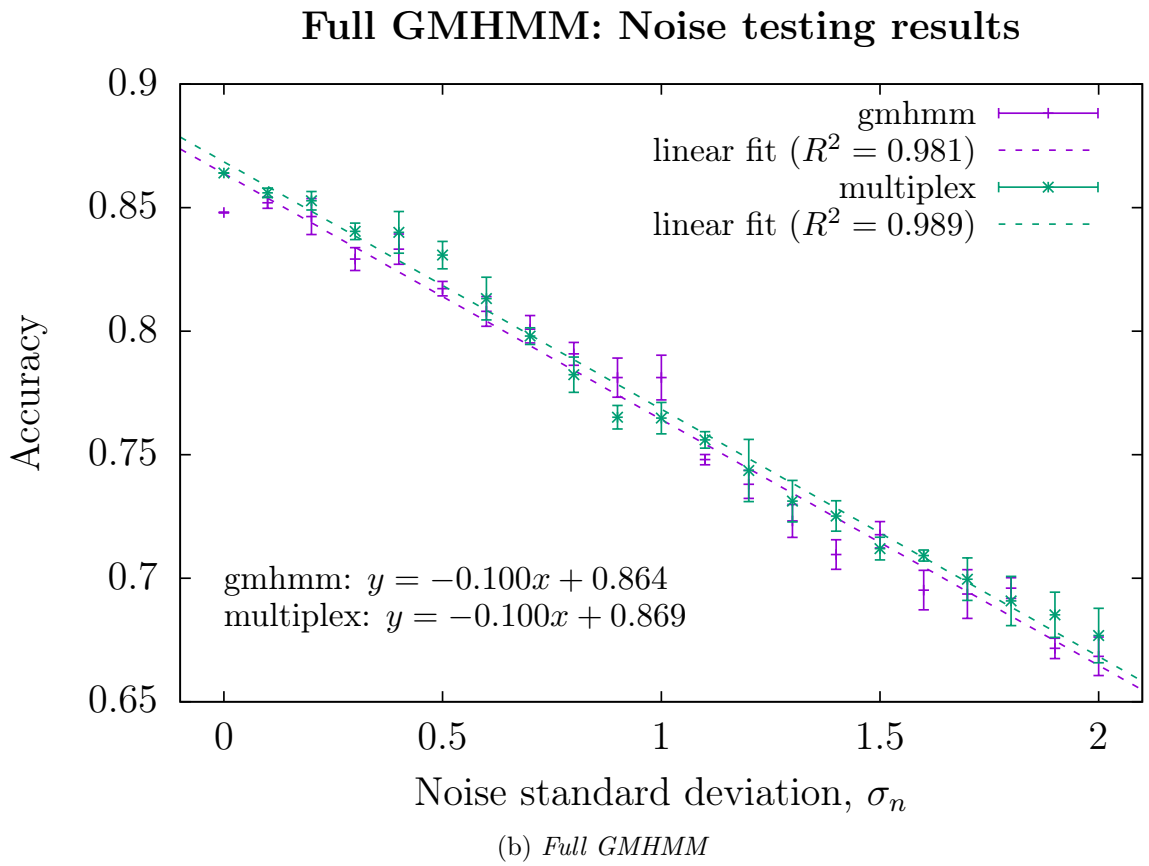
The results of the analysis are presented in Figure 4.5. These results successfully demonstrate that the classification accuracy of the four implemented models decreases *linearly* with the increase in the noise standard deviation, and as such, they demonstrate a **good level of robustness to noise**. Furthermore, the comparative linear fits demonstrate that introducing multiplexity into the single-layer models does not cause any dramatic changes in robustness (the slopes of the linear fits remaining around -0.1).

4.5 Summary

In this chapter, I have provided a review of the methods used to verify that the project has met its success criteria, and the actual results obtained by those methods. This involved a brief overview of the adopted unit testing strategy, followed by a presentation of the obtained comparative evaluation and robustness analysis results.



(a) *Chain GMHMM*



(b) *Full GMHMM*

Figure 4.5: Results of noise testing on the implemented models; error bars represent the standard error of the sample mean.

Chapter 5

Conclusions

Within this dissertation I have provided a thorough documentation of the preparatory, implementation and evaluation work I have performed while developing my project.

This section will contain a few concluding remarks: a final summary of what has been achieved, the lessons I have learnt after successfully executing a project of this scale, and an overview of potential further extensions to the project.

5.1 Results

As discussed in the previous chapters, **the project has met and exceeded all of its success criteria**, producing all of the expected core implementation deliverables along with several extensions. The models and their multiplex counterparts have both shown impressive performance and robustness when faced with a biomolecular classification problem on real data sets, and I have also demonstrated that introducing multiplexity results in more powerful classifiers without sacrificing robustness.

Complex networks are currently a very active area of research; however, thus far little attention has been given to their utilisation in machine learning problems. I'm hopeful that the successes of my project—especially given that, to the best of my knowledge, there are no freely available implementations of multiplex networks in machine learning contexts—will provide a stepping-stone for me to partake in actively researching this application, as well as complex networks in general.

5.2 Lessons learnt

While not a research project by nature, my project involved combining known models and algorithms in a previously completely *unexplored* context. This brought with it a great amount of risk, and a number of moments when there was no 'prescribed' way of integrating the implemented components. Successfully overcoming issues like those required hours of research into ways that similar problems have been previously solved, and adapting the methods to be suitable within the context of my implementation.

To me, this represented a very suitable introduction to methods of conducting research, and should be an important foundational skill for further study.

5.3 Further work

Although the project has been successful in achieving its success criteria, during development I have managed to identify several aspects where the implementation could be refined; these are summarised below.

- *Most-likely sequences of states.* The Baum-Welch algorithm produces a network where the states do not necessarily have any known meaning attached to them. Perhaps a way into researching these meanings would be to apply the Viterbi algorithm (Appendix A.2) to the models and analyse the obtained state sequences; this would likely require collaboration with experts in the relevant fields.

- *Optimising the NSGA-II parameters.* The parameters I have utilised for the genetic algorithm are the same as suggested in Deb *et al.* [15]; the authors of the paper have explicitly stated that they have made no effort to optimise these parameters. While I am very satisfied with the performance shown by my model currently, it would be a viable effort to investigate how these parameters could be optimised, in order to obtain better quality solutions, higher rates of convergence, etc.

- *Parallelisation.* Several components within the model implementation as well as the evaluation suite are *'embarrassingly parallelisable'*, meaning that entire routines could easily be placed in separate threads to speed up the execution. Simple examples of this are the individual sequence trainings within the Baum-Welch algorithm and the crossover routine within NSGA-II. I am hoping to do a more thorough investigation of this and produce a model that executes faster through the use of concurrency.

Bibliography

[1] Malbor Asllani, Daniel M Busiello, Timoteo Carletti, Duccio Fanelli, and Gwendoline Planchon. Turing patterns in multiplex networks. *Physical Review E*, 90(4):042814, 2014.

[2] N Azimi-Tafreshi, J Gómez-Gardeñes, and SN Dorogovtsev. k- core percolation on multiplex networks. *Physical Review E*, 90(3):032816, 2014.

[3] Thomas Back. *Evolutionary algorithms in theory and practice*. Oxford Univ. Press, 1996.

[4] Leonard E Baum and Ted Petrie. Statistical inference for probabilistic functions of finite state markov chains. *The annals of mathematical statistics*, pages 1554–1563, 1966.

[5] Leonard E Baum, Ted Petrie, George Soules, and Norman Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains. *The annals of mathematical statistics*, pages 164–171, 1970.

[6] Ginestra Bianconi. Statistical mechanics of multiplex networks: Entropy and overlap. *Physical Review E*, 87(6):062806, 2013.

[7] Béla Bollobás. *Modern graph theory*, volume 184. Springer Science & Business Media, 1998.

[8] Alessio Cardillo, Jesús Gómez-Gardeñes, Massimiliano Zanin, Miguel Romance, David Papo, Francisco Del Pozo, and Stefano Boccaletti. Emergence of network features from multiplexity. *Scientific reports*, 3, 2013.

[9] Manlio De Domenico, Andrea Lancichinetti, Alex Arenas, and Martin Rosvall. Identifying modular flows on multilayer networks reveals highly overlapping organization in social systems. *arXiv preprint arXiv:1408.2925*, 2014.

[10] Manlio De Domenico, Antonio Lima, Paul Mougel, and Mirco Musolesi. The anatomy of a scientific rumor. *Scientific reports*, 3, 2013.

[11] Manlio De Domenico, Mason A Porter, and Alex Arenas. Multilayer analysis and visualization of networks. *arXiv preprint arXiv:1405.0843*, 2014.

[12] Manlio De Domenico, Albert Solé-Ribalta, Sergio Gómez, and Alex Arenas. Navigability of interconnected networks under random failures. *Proceedings of the National Academy of Sciences*, 111(23):8351–8356, 2014.

[13] Kalyanmoy Deb. *Multi-objective optimization using evolutionary algorithms*, volume 16. John Wiley & Sons, 2001.

[14] Kalyanmoy Deb and Ram B Agrawal. Simulated binary crossover for continuous search space. *Complex Systems*, 9(3):1–15, 1994.

[15] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Evolutionary Computation, IEEE Transactions on*, 6(2):182–197, 2002.

[16] 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)*, pages 1–38, 1977.

[17] Ernesto Estrada and Jesús Gómez-Gardeñes. Communicability reveals a transition to coordinated behavior in multiplex networks. *Physical Review E*, 89(4):042819, 2014.

[18] JD Ferguson. Hidden markov analysis: an introduction. In *Proceedings of the Symposium on the Applications of Hidden Markov Models to Text and Speech, IDA-CRD, Princeton, NJ*, 1980.

[19] Emilio Frazzoli. Mit ocw 16.410/413 - principles of autonomy and decision making, fall 2010. *MIT OpenCourseWare: Massachusetts Institute of Technology*, 2010.

[20] Clara Granell, Sergio Gómez, and Alex Arenas. Competing spreading processes on multiplex networks: awareness and epidemics. *Physical Review E*, 90(1):012808, 2014.

[21] Desislava Hristova, Mirco Musolesi, and Cecilia Mascolo. Keep your friends close and your facebook friends closer: A multiplex network approach to the analysis of offline and online social ties. *CoRR*, abs/1403.8034, 2014.

[22] Yanqing Hu, Shlomo Havlin, and Hernán A Makse. Conditions for viral influence spreading through multiplex correlated social networks. *Physical Review X*, 4(2):021031, 2014.

[23] Mikko Kivelä, Alex Arenas, Marc Barthelemy, James P Gleeson, Yamir Moreno, and Mason A Porter. Multilayer networks. *Journal of Complex Networks*, 2(3):203–271, 2014.

[24] David Krackhardt. Cognitive social structures. *Social networks*, 9(2):109–134, 1987.

[25] Chuan Wen Loe and Henrik Jeldtoft Jensen. Comparison of communities detection algorithms for multiplex. *arXiv preprint arXiv:1406.2205*, 2014.

[26] Matteo Magnani, Barbora Micenkova, and Luca Rossi. Combinatorial analysis of multiple networks. *arXiv preprint arXiv:1303.4986*, 2013.

[27] Марков, АА. Распространение закона больших чисел на величины, зависящие друг от друга. *Известия Физико-математического общества при Казанском университете*, pages 135–156, 1906.

[28] Brian W Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA)-Protein Structure*, 405(2):442–451, 1975.

[29] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2012.

[30] Marc Mouret, Christine Solnon, and Christian Wolf. Classification of images based on hidden markov models. In *Content-Based Multimedia Indexing, 2009. CBMI'09. Seventh International Workshop on*, pages 169–174. IEEE, 2009.

[31] Nabil Mohammed Ali Munassar and A Govardhan. A comparison between five models of software engineering. *IJCSI*, 5:95–101, 2010.

[32] Vincenzo Nicosia, Ginestra Bianconi, Vito Latora, and Marc Barthelemy. Growing multiplex networks. *Physical review letters*, 111(5):058701, 2013.

[33] James R Norris. *Markov chains*. Number 2. Cambridge university press, 1998.

[34] John F Padgett and Christopher K Ansell. Robust action and the rise of the medici, 1400-1434. *American journal of sociology*, pages 1259–1319, 1993.

[35] Bryan Pardo and William Birmingham. Modeling form for on-line following of musical performances. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 20, page 1018. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2005.

[36] David Martin Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. 2011.

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

[38] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[39] L Satish and BI Gururaj. Use of hidden markov models for partial discharge pattern classification. *Electrical Insulation, IEEE Transactions on*, 28(2):172–182, 1993.

[40] Chris Stark, Bobby-Joe Breitkreutz, Teresa Reguly, Lorrie Boucher, Ashton Breitkreutz, and Mike Tyers. Biogrid: a general repository for interaction datasets. *Nucleic acids research*, 34(suppl 1):D535–D539, 2006.

[41] Thad Starner and Alex Pentland. Real-time american sign language visual recognition from video using hidden markov models. *Master's Thesis, MIT Program in Media Arts*, 1995.

[42] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

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

[44] Dawei Zhao, Lixiang Li, Haipeng Peng, Qun Luo, and Yixian Yang. Multiple routes transmitted epidemics on multiplex networks. *Physics Letters A*, 378(10):770–776, 2014.

Appendix A

Further theory

A.1 Introduction to hidden Markov models

A.1.1 Markov chains

One simple way of defining a (discrete-state, discrete-time[1]) hidden Markov model is by augmenting its simpler variant, the **Markov chain** [27]. Both of these models are stochastic models that satisfy the **Markov property** (*memorylessness*):

Definition 4. Let S be a countable set of *states*, and $\{X_n\}_{n\geq 0}$ a sequence of discrete random variables taking values $\{x_n\}_{n\geq 0}$ s.t. $\forall n \geq 0.\ x_n \in S$. This sequence satisfies the *Markov property* if

$$\forall n \geq 1.\ \mathbb{P}(X_n = x_n | X_{n-1} = x_{n-1}, \ldots, X_0 = x_0) = \mathbb{P}(X_n = x_n | X_{n-1} = x_{n-1})$$

and it is then called a *Markov chain*.

Definition 4 specifies that the probability distribution of the next state a Markov chain will have is determined solely by its current state. Furthermore it is often assumed that this distribution doesn't change over time (**time-homogeneity**):

Definition 5. A Markov chain $\{X_n\}_{n\geq 0}$ is *time-homogeneous* if

$$\forall n \geq 1.\ \forall x, y \in S.\ \mathbb{P}(X_n = y | X_{n-1} = x) = \mathbb{P}(X_1 = y | X_0 = x)$$

Assuming time-homogeneity is convenient for representing Markov chains; if we also assume that the state set S is finite, to fully specify the dynamics of any time-homogeneous Markov chain it is sufficient to specify:

- A **start-state probability vector**[2], $\vec{\pi}$, defined such that $\pi_x \stackrel{\text{def}}{=} \mathbb{P}(X_0 = x)$;
- A **transition matrix**, $\mathbf{T}$, defined such that $\mathbf{T}_{xy} \stackrel{\text{def}}{=} \mathbb{P}(X_1 = y | X_0 = x)$.

[1] It is possible to extend the definitions to the continuous case [33], however this is outside the scope of my project, and is omitted.

[2] The start-state probability vector may be omitted if the start state is known in advance.

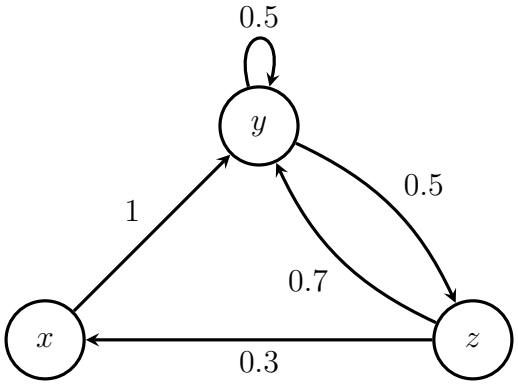


Figure A.1: Example of a time-homogeneous Markov chain with the state set $S = \{x, y, z\}$.

The probability of observing any particular state sequence $\{x_t\}_{t=0}^{T}$ then amounts to

$$\mathbb{P}(\{x_t\}_{t=0}^{T}) = \mathbb{P}(X_0 = x_0) \prod_{t=1}^{T} \mathbb{P}(X_t = x_t | X_{t-1} = x_{t-1}) = \pi_{x_0} \prod_{t=0}^{T-1} \mathbf{T}_{x_t x_{t+1}}$$

In addition, such a Markov chain can be simply represented as a probabilistic finite state machine (refer to Figure A.1).

A.1.2 Hidden Markov models

A hidden Markov model can now be built from the Markov chain, by means of the following definition:

Definition 6. A *hidden Markov model* (HMM) is a Markov chain in which the state sequence may be unobservable (*hidden*).

This means that, while the Markov chain parameters (e.g. transition matrix and start-state probabilities in the time-homogeneous case) are still known, there is no way to determine the state sequence $\{X_n\}_{n\geq 0}$ the system will follow. What can be observed is the **output** sequence produced, $\{Y_n\}_{n\geq 0}$. The output sequence can assume any value from a given **set of outputs**, O. It is assumed that the output at any given moment **depends only on the current state**:

$$\forall n \geq 0.\, \mathbb{P}(Y_n = y_n | X_n = x_n, \ldots, X_0 = x_0, Y_{n-1} = y_{n-1}, \ldots, Y_0 = y_0) = \mathbb{P}(Y_n = y_n | X_n = x_n)$$

If we furthermore assume that the set of outputs is finite, and that the output probability distribution doesn't change with time, i.e.

$$\forall n \geq 0.\, \mathbb{P}(Y_n = y_n | X_n = x_n) = \mathbb{P}(Y_0 = y_n | X_0 = x_n)$$

then the only additional parameter we need to fully specify an HMM is the **output probability matrix**, $\mathbf{O}$, defined by $\mathbf{O}_{xy} \stackrel{\text{def}}{=} \mathbb{P}(Y_0 = y | X_0 = x)$.

An example of an HMM (extended from the Markov chain example) is given in Figure A.2.

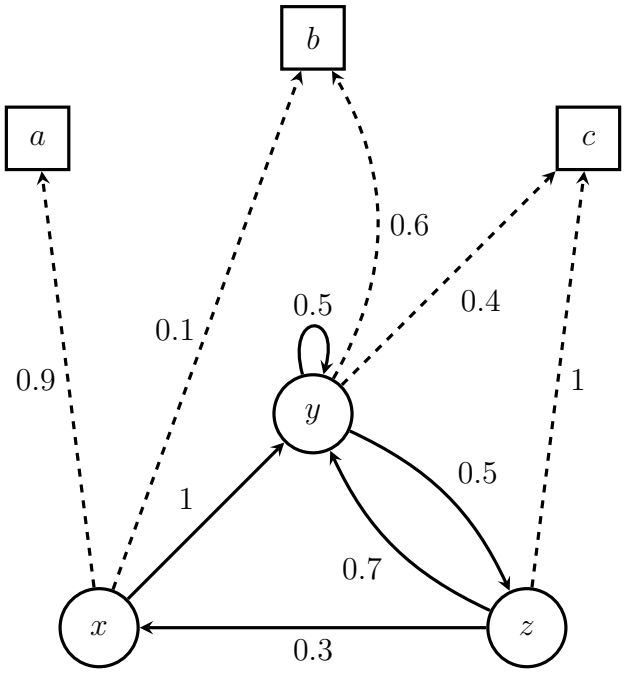


Figure A.2: Example of a hidden Markov model, extended from the Markov chain given in Figure A.1 with the set of outputs $O = \{a, b, c\}$.

A.1.3 Learning and inference

As previously mentioned, an HMM is capable of solving a variety of machine learning problems; the fundamental three problems (as given in [18]) are summarised here. To denote the HMM model parameters when describing the problems, the notation $\mathbf{\Theta} \stackrel{\text{def}}{=} (\vec{\pi}, \mathbf{T}, \mathbf{O})$ will be used.

- *Probability of an observed output sequence.* Given an output sequence, $\{y_t\}_{t=0}^{T}$, determine the probability that it was produced by the given HMM $\mathbf{\Theta}$, i.e.

$$\mathbb{P}(Y_0 = y_0, \ldots, Y_T = y_T | \mathbf{\Theta}) \tag{A.1}$$

  This problem is efficiently solved with the **forward algorithm** (§3.3.2.2).

- *Most likely sequence of states for an observed output sequence.* Given an output sequence, $\{y_t\}_{t=0}^{T}$, determine the most likely sequence of states, $\{\widehat{x}_t\}_{t=0}^{T}$, that produced it within a given HMM $\mathbf{\Theta}$, i.e.

$$\{\widehat{x}_t\}_{t=0}^{T} \stackrel{\text{def}}{=} \underset{\{x_t\}_{t=0}^{T}}{\operatorname{argmax}} \ \mathbb{P}(\{x_t\}_{t=0}^{T} | \{y_t\}_{t=0}^{T}, \mathbf{\Theta}) \tag{A.2}$$

  This problem is efficiently solved with the **Viterbi algorithm** [43] (Appendix A.2[3]).

- *Adjusting the model parameters.* Given an output sequence, $\{y_t\}_{t=0}^{T}$ and an HMM $\mathbf{\Theta}$, produce a new HMM $\mathbf{\Theta}'$ that is more likely to produce that sequence, i.e.

$$\mathbb{P}(\{y_t\}_{t=0}^{T} | \mathbf{\Theta}') \geq \mathbb{P}(\{y_t\}_{t=0}^{T} | \mathbf{\Theta}) \tag{A.3}$$

  This problem is efficiently solved with the **Baum-Welch Algorithm** (§3.4.2).

[3]The Viterbi algorithm is not used within my project, however I am describing it within an appendix because it could reasonably be applied in future work.

A.2 Viterbi algorithm

This appendix will contain a brief description of the Viterbi algorithm [43], which aims to find the most likely state sequence that produced a given output sequence $\vec{y}$ in a given HMM $\mathbf{\Theta} = (\vec{\pi}, \mathbf{T}, \mathbf{O})$.

The algorithm employs a standard dynamic programming approach, computing the overall likelihood of ending up in state x after processing the first t elements of the output, $V_t(x)$. The base cases, $V_1(x)$, are simply calculated as

$$V_1(x) = \vec{\pi}_x \mathbf{O}_{x,y_1}$$

corresponding to starting in state x and producing the first output from it. After computing all $V_t(x)$ values for some t, computing $V_{t+1}(x)$ may proceed as follows:

$$V_{t+1}(x) = \max_{x' \in S} V_t(x') \mathbf{T}_{x'x} \mathbf{O}_{x,y_{t+1}}$$

corresponding to ending up in x' in the previous point in time, transitioning from x' to x, and then producing the $(t+1)$-st output from it. It is also useful to store which state x' has optimised the likelihood:

$$Ptr_{t+1}(x) = \operatorname*{argmax}_{x' \in S} V_t(x') \mathbf{T}_{x'x} \mathbf{O}_{x,y_{t+1}}$$

The optimal state sequence $\vec{x}^*$, is then computed by backtracking; the final state x^*_T is simply the one which maximises the overall likelihood, i.e.

$$x^*_T = \operatorname*{argmax}_{x \in S} V_T(x)$$

and further states may be obtained using the relation

$$x^*_t = Ptr_{t+1}(x^*_{t+1})$$

As with the majority of the algorithms discussed within this dissertation, the Viterbi algorithm is also prone to underflow problems; luckily, as all computations involved are multiplications, it is possible to trivially take the logarithm, and work only with sums of logarithms while computing the entries of $V_t(x)$.

A.3 Simulated binary crossover/polynomial mutation

Within this appendix, I will state the formulae for the simulated binary crossover *(SBX)* operator, to produce two children real numbers p' and q' from parents p and q. Furthermore, I will provide the formulae for the polynomial mutation operator, to mutate a given real number x, producing a new real number x'.

The SBX operator has been defined in [14], and attempts to simulate the single-point crossover operator for binary strings; choosing a single point in the string after

which the strings' contents are swapped to produce the children. Two parameters need to be specified: a real number $u \in [0, 1]$ drawn uniformly at random, and a parameter η_c (used to control the spread of the produced solutions). First, define β as follows:

$$\beta \stackrel{\text{def}}{=} \begin{cases} (2u)^{\frac{1}{\eta_c+1}} & u \leq 0.5 \\ \left(\frac{1}{2(1-u)}\right)^{\frac{1}{\eta_c+1}} & \text{otherwise} \end{cases}$$

Then the produced children are:

$$p' = 0.5((1+\beta)p + (1-\beta)q)$$

$$q' = 0.5((1-\beta)p + (1+\beta)q)$$

Polynomial mutation attempts to introduce a variation in the value present; it does so by sampling a polynomial distribution. Similarly as before, a real number $u \in [0, 1]$ drawn uniformly at random is needed, as well as a spread-controlling parameter η_m. Define δ as follows:

$$\delta \stackrel{\text{def}}{=} \begin{cases} (2u)^{\frac{1}{\eta_m+1}} - 1 & u \leq 0.5 \\ 1 - (2(1-u))^{\frac{1}{\eta_m+1}} & \text{otherwise} \end{cases}$$

Then the mutated value is:

$$x' = x + (x_{hi} - x_{lo})\delta$$

where x_{lo} and x_{hi} are the lower and upper bounds on x, respectively (constrained by the problem to be solved).

Appendix B

Code samples

This appendix contains select portions of the C++ code contained within my project's codebase, mainly for the purpose of illustrating some of the algorithms presented in the Implementation chapter.

B.1 NSGA-II

B.1.1 Fast nondominated sort

```
vector<vector<chromosome> > NSGAII::fast_nondominated_sort(vector<
    chromosome> &P)
{
  vector<vector<chromosome> > F;

  vector<vector<int> > Sp;
  vector<int> np, Q;
  Sp.resize(P.size());
  np.resize(P.size());
  for (uint i=0;i<P.size();i++)
  {
    chromosome p = P[i];
    Sp[i].clear();
    np[i] = 0;
    for (uint j=0;j<P.size();j++)
    {
      chromosome q = P[j];
      if (dominated_by(p, q)) Sp[i].push_back(j);
      else if (dominated_by(q, p)) np[i]++;
    }
    if (np[i] == 0) Q.push_back(i);
  }

  vector<chromosome> Fi;
  Fi.resize(Q.size());
  for (uint i=0;i<Q.size();i++)
  {
    Fi[i] = P[Q[i]];
```

```
28      Fi[i].rank = 1;
29    }
30    F.push_back(Fi);
31
32    int ii = 1;
33    while (!Q.empty())
34    {
35      vector<int> R;
36      for (uint i=0;i<Q.size();i++)
37      {
38        for (uint j=0;j<Sp[Q[i]].size();j++)
39        {
40          int q = Sp[Q[i]][j];
41          if (--np[q] == 0) R.push_back(q);
42        }
43      }
44      ii++;
45      if (R.empty()) break;
46
47      Fi.resize(R.size());
48      for (uint i=0;i<R.size();i++)
49      {
50        Fi[i] = P[R[i]];
51        Fi[i].rank = ii;
52      }
53      F.push_back(Fi);
54      Q = R;
55    }
56
57    P.clear();
58    return F;
59  }
```

B.1.2 Crowding distance assignment

```
1  void NSGAII::crowding_distance_assignment(vector<chromosome> &I)
2  {
3    int l = I.size();
4    for (int i=0;i<l;i++) I[i].distance = 0;
5    for (int obj=0;obj<obj_size;obj++)
6    {
7      for (int i=0;i<l;i++) I[i].sort_key = obj;
8      sort(I.begin(), I.end(), cmp_by_key);
9      I[0].distance = I[l-1].distance = INF;
10     for (int i=1;i<l-1;i++)
11     {
12       I[i].distance += I[i+1].values[obj] - I[i-1].values[obj];
13     }
14   }
15 }
```

B.1.3 Solution combining

```
void NSGAII::make_new_pop(vector<chromosome> &P)
{
  for (int i=0;i<pop_size >> 1;i++)
  {
    chromosome p1, p2;
    p1 = P[select(P)];
    p2 = P[select(P)];
    pair<chromosome, chromosome> Cret = crossover(p1, p2);
    P.push_back(Cret.first);
    P.push_back(Cret.second);
  }
  mutate(P); // mutates only new generation, leaves first N alone
  for (int i=0;i<pop_size;i++)
  {
    for (int obj=0;obj<obj_size;obj++)
    {
      P[pop_size + i].values[obj] = objectives[obj](P[pop_size + i].
   features);
    }
  }
}
```

B.1.4 Full iteration

```
void NSGAII::iterate()
{
  make_new_pop(main_population);
  vector<vector<chromosome> > fronts = fast_nondominated_sort(
   main_population);
  int ii = 0;
  while (main_population.size() + fronts[ii].size() <= uint(pop_size))
  {
    crowding_distance_assignment(fronts[ii]);
    main_population.insert(main_population.end(), fronts[ii].begin(),
   fronts[ii].end());
    ii++;
  }
  int elements_needed = pop_size - main_population.size();
  if (elements_needed > 0)
  {
    crowding_distance_assignment(fronts[ii]);
    sort(fronts[ii].begin(), fronts[ii].end());
    main_population.insert(main_population.end(), fronts[ii].begin(),
   fronts[ii].begin() + elements_needed);
  }
}
```

B.2 GMHMM

B.2.1 Forward algorithm

```
tuple<double**, double*, double> GMHMM::forward(vector<pair<double, int>
    > &Y)
{
  int Ti = Y.size();

  double **alpha = new double*[Ti];
  for (int i=0;i<Ti;i++)
  {
    alpha[i] = new double[n];
  }
  double *c = new double[Ti];

  double sum = 0.0;
  for (int i=0;i<n;i++)
  {
    alpha[0][i] = pi[i] * O[i][Y[0].second] * get_probability(Y[0].
   second, Y[0].first);
    sum += alpha[0][i];
  }
  c[0] = 1.0 / sum;
  for (int i=0;i<n;i++)
  {
    alpha[0][i] /= sum;
  }

  for (int t=1;t<Ti;t++)
  {
    sum = 0.0;
    for (int i=0;i<n;i++)
    {
      alpha[t][i] = 0.0;
      for (int j=0;j<n;j++)
      {
        alpha[t][i] += alpha[t-1][j] * T[j][i];
      }
      alpha[t][i] *= O[i][Y[t].second] * get_probability(Y[t].second, Y[
   t].first);
      sum += alpha[t][i];
    }

    c[t] = 1.0 / sum;
    for (int i=0;i<n;i++)
    {
      alpha[t][i] /= sum;
    }
  }

  double log_L = 0.0;
  for (int i=0;i<Ti;i++) log_L -= log(c[i]);
```

```
47
48    return make_tuple(alpha, c, log_L);
49  }
```

B.2.2 Baum-Welch algorithm

```
1   void GMHMM::baumwelch(vector<vector<double> > &Ys, int iterations,
        double tolerance)
2   {
3     vector<vector<pair<double, int> > > sorted_Ys;
4     sorted_Ys.resize(Ys.size());
5
6     for (uint l=0;l<Ys.size();l++)
7     {
8       sorted_Ys[l].resize(Ys[l].size());
9       for (uint i=0;i<Ys[l].size();i++) sorted_Ys[l][i] = make_pair(Ys[l][
        i], i);
10      sort(sorted_Ys[l].begin(), sorted_Ys[l].end());
11    }
12
13    double ***alpha = new double**[sorted_Ys.size()];
14    double ***beta = new double**[sorted_Ys.size()];
15    double **c = new double*[sorted_Ys.size()];
16
17    double PP, QQ;
18
19    double lhood = 0.0;
20    double oldlhood = 0.0;
21
22    for (int iter=0;iter<iterations;iter++)
23    {
24      lhood = 0.0;
25
26      for (uint l=0;l<sorted_Ys.size();l++)
27      {
28        tuple<double**, double*, double> x = forward(sorted_Ys[l]);
29        alpha[l] = get<0>(x);
30        c[l] = get<1>(x);
31        lhood += get<2>(x);
32        beta[l] = backward(sorted_Ys[l], c[l]);
33      }
34
35      double **next0 = new double*[n];
36      for (int i=0;i<n;i++) next0[i] = new double[obs];
37
38      for (int i=0;i<n;i++)
39      {
40        pi[i] = 0.0;
41        for (uint l=0;l<sorted_Ys.size();l++)
42        {
43          pi[i] += alpha[l][0][i] * beta[l][0][i];
44        }
```

```
45        pi[i] /= sorted_Ys.size();
46
47        QQ = 0.0;
48
49        for (int k=0;k<obs;k++)
50        {
51          nextO[i][k] = 0.0;
52        }
53
54        for (uint l=0;l<sorted_Ys.size();l++)
55        {
56          for (uint t=0;t<sorted_Ys[l].size()-1;t++)
57          {
58            double curr = alpha[l][t][i] * beta[l][t][i];
59            QQ += curr;
60            nextO[i][sorted_Ys[l][t].second] += curr;
61          }
62        }
63
64        for (int j=0;j<n;j++)
65        {
66          PP = 0.0;
67          for (uint l=0;l<sorted_Ys.size();l++)
68          {
69            for (uint t=0;t<sorted_Ys[l].size()-1;t++)
70            {
71              PP += alpha[l][t][i] * O[j][sorted_Ys[l][t+1].second] *
    get_probability(sorted_Ys[l][t+1].second, sorted_Ys[l][t+1].first) *
    beta[l][t+1][j] * c[l][t+1];
72            }
73          }
74          T[i][j] *= PP / QQ;
75        }
76
77        for (uint l=0;l<sorted_Ys.size();l++)
78        {
79          int lim = sorted_Ys[l].size() - 1;
80          double curr = alpha[l][lim][i] * beta[l][lim][i];
81          QQ += curr;
82          nextO[i][sorted_Ys[l][lim].second] += curr;
83        }
84
85        for (int k=0;k<obs;k++)
86        {
87          nextO[i][k] /= QQ;
88        }
89      }
90
91      for (uint l=0;l<sorted_Ys.size();l++)
92      {
93        for (uint t=0;t<sorted_Ys[l].size();t++)
94        {
95          delete[] alpha[l][t];
```

```
96          delete[] beta[l][t];
97        }
98        delete[] alpha[l];
99        delete[] beta[l];
100       delete[] c[l];
101     }
102
103     for (int i=0;i<n;i++)
104     {
105       delete[] O[i];
106     }
107     delete[] O;
108
109     O = nextO;
110
111     if (fabs(lhood - oldlhood) < tolerance) break;
112     oldlhood = lhood;
113   }
114 }
```

B.2.3 Multiplex training

```
1  void MultiplexGMHMM::train(vector<vector<vector<double> > > &train_set)
2  {
3    // Train all the layers individually (as before)
4    for (int l=0;l<L;l++)
5    {
6      vector<vector<double> > curr_set(train_set.size(), vector<double>(
       obs));
7      for (uint i=0;i<train_set.size();i++)
8      {
9        for (int j=0;j<obs;j++)
10       {
11         curr_set[i][j] = train_set[i][j][l];
12       }
13     }
14     layers[l] -> train(curr_set);
15   }
16
17   // Run the algorithm
18   NSGAII nsga2;
19   vector<chromosome> candidates = nsga2.optimise(params, objectives);
20
21   // Evaluate the best choice of omega
22   int best = -1;
23   double min_sum = -1.0;
24   for (uint i=0;i<candidates.size();i++)
25   {
26     sort(candidates[i].values.begin(), candidates[i].values.end());
27     double curr_sum = 0.0;
28     for (uint j=0;j<train_set.size();j++)
29     {
```

```
30        curr_sum += (j + 1) * candidates[i].values[j];
31      }
32      if (best == -1 || curr_sum < min_sum)
33      {
34        best = i;
35        min_sum = curr_sum;
36      }
37    }
38
39    // Adjust the parameters accordingly
40    double **fin_omega = new double*[L];
41    for (int i=0;i<L;i++)
42    {
43      fin_omega[i] = new double[L];
44      for (int j=0;j<L;j++)
45      {
46        fin_omega[i][j] = candidates[best].features[i*L + j];
47      }
48    }
49
50    set_omega(fin_omega);
51
52    for (int i=0;i<L;i++) delete[] fin_omega[i];
53    delete[] fin_omega;
54  }
```

Appendix C

Unit tests

This appendix contains a direct reference to the unit tests used to verify correctness of the principal algorithms within my project.

It should be noted that the HMM algorithms have been tested on an HMM with *discrete* outputs, due to the greater abundance of such examples available. This is sufficient, due to the fact that extending them to the GMHMM case only requires multiplying by an additional value (the PDF of a Gaussian distribution).

C.1 Forward/backward algorithm

This test case is based on the "umbrella world", given in Russell & Norvig [38].

The parameters of the HMM are:

$$n = m = 2$$

$$\vec{\pi} = \begin{pmatrix} 0.5 & 0.5 \end{pmatrix}$$

$$\mathbf{T} = \begin{pmatrix} 0.7 & 0.3 \\ 0.3 & 0.7 \end{pmatrix}$$

$$\mathbf{O} = \begin{pmatrix} 0.9 & 0.1 \\ 0.2 & 0.8 \end{pmatrix}$$

The observed output sequence is:

$$\vec{y} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

The expected (normalised) forward and backward likelihoods after each step are:

$$\hat{\alpha} = \begin{pmatrix} 0.8182 & 0.8834 & 0.1907 & 0.7308 & 0.8673 \\ 0.1818 & 0.1166 & 0.8093 & 0.2692 & 0.1327 \end{pmatrix}$$

$$\hat{\beta} = \begin{pmatrix} 0.5923 & 0.3763 & 0.6533 & 0.6273 & 0.5000 \\ 0.4077 & 0.6237 & 0.3467 & 0.3727 & 0.5000 \end{pmatrix}$$

C.2 Baum-Welch algorithm

This test case is based on the "Finding Keyser Söze" worked example given in a lecture by E. Frazzoli [19].

The initial HMM parameters are:

$$n = 2, m = 3$$

$$\vec{\pi} = \begin{pmatrix} 0.5 & 0.5 \end{pmatrix}$$

$$\mathbf{T} = \begin{pmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{pmatrix}$$

$$\mathbf{O} = \begin{pmatrix} 0.4 & 0.1 & 0.5 \\ 0.1 & 0.5 & 0.4 \end{pmatrix}$$

The observed output sequence is:

$$\vec{y} = \begin{pmatrix} 2 & 0 & 0 & 2 & 1 & 2 & 1 & 1 & 1 & 2 & 1 & 1 & 1 & 1 & 1 & 2 & 2 & 0 & 0 & 1 \end{pmatrix}$$

After convergence, the following parameters of the HMM are expected[1]:

$$\vec{\pi}' = \begin{pmatrix} 1 & 0 \end{pmatrix}$$

$$\mathbf{T}' = \begin{pmatrix} 0.6909 & 0.3091 \\ 0.0934 & 0.9066 \end{pmatrix}$$

$$\mathbf{O}' = \begin{pmatrix} 0.5807 & 0.0010 & 0.4183 \\ 0.0000 & 0.7621 & 0.2379 \end{pmatrix}$$

C.3 NSGA-II

The test cases used are the same as the ones used in Deb *et al.* [15] to evaluate NSGA-II, and are outlined in Table C.1.

[1] As termination criteria may be different, the final output is accepted if it is within $\varepsilon = 10^{-3}$ of the expected parameters.

Test	n	Constraints	Objectives (to minimise)	Optimal front
SCH	1	$x \in [-10^3, 10^3]$	$f_1(x) = x^2$ $f_2(x) = (x-2)^2$	$x \in [0, 2]$
FON	3	$x_i \in [-4, 4]$	$f_1(\vec{x}) = 1 - \exp\left(-\sum_{i=1}^{3}\left(x_i - \frac{1}{\sqrt{3}}\right)^2\right)$ $f_2(\vec{x}) = 1 - \exp\left(-\sum_{i=1}^{3}\left(x_i + \frac{1}{\sqrt{3}}\right)^2\right)$	$x_1 = x_2 = x_3$ $\in [-\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}]$
POL	2	$x_i \in [-\pi, \pi]$	$f_1(\vec{x}) = 1 + (A_1 - B_1)^2 + (A_2 - B_2)^2$ $f_2(\vec{x}) = (x_1+3)^2 + (x_2+1)^2$ $A_1 = 0.5\sin 1 - 2\cos 1 + \sin 2 - 1.5\cos 2$ $A_2 = 1.5\sin 1 - \cos 1 + 2\sin 2 - 0.5\cos 2$ $B_1 = 0.5\sin x_1 - 2\cos x_1 + \sin x_2 - 1.5\cos x_2$ $B_2 = 1.5\sin x_1 - \cos x_1 + 2\sin x_2 - 0.5\cos x_2$	ref. [13]
KUR	3	$x_i \in [-5, 5]$	$f_1(\vec{x}) = \sum_{i=1}^{n-1} -10\exp\left(-0.2\sqrt{x_i^2 + x_{i+1}^2}\right)$ $f_2(\vec{x}) = \sum_{i=1}^{n} \lvert x_i\rvert^{0.8} + 5\sin x_i^3$	ref. [13]
ZDT1	30	$x_i \in [0, 1]$	$f_1(\vec{x}) = x_1$ $f_2(\vec{x}) = g(\vec{x})\left(1 - \sqrt{\frac{x_1}{g(\vec{x})}}\right)$ $g(\vec{x}) = 1 + 9\frac{\sum_{i=2}^{n} x_i}{n-1}$	$x_1 \in [0, 1]$ $x_i = 0$ $(i \neq 1)$
ZDT2	30	$x_i \in [0, 1]$	$f_1(\vec{x}) = x_1$ $f_2(\vec{x}) = g(\vec{x})\left(1 - \left(\frac{x_1}{g(\vec{x})}\right)^2\right)$ $g(\vec{x}) = 1 + 9\frac{\sum_{i=2}^{n} x_i}{n-1}$	$x_1 \in [0, 1]$ $x_i = 0$ $(i \neq 1)$
ZDT3	30	$x_i \in [0, 1]$	$f_1(\vec{x}) = x_1$ $f_2(\vec{x}) = g(\vec{x})\left(1 - \sqrt{\frac{x_1}{g(\vec{x})}} - \frac{x_1}{g(\vec{x})}\sin(10\pi x_1)\right)$ $g(\vec{x}) = 1 + 9\frac{\sum_{i=2}^{n} x_i}{n-1}$	$x_1 \in [0, 1]$ $x_i = 0$ $(i \neq 1)$
ZDT4	10	$x_1 \in [0, 1]$ $x_i \in [-5, 5]$ $(i \neq 1)$	$f_1(\vec{x}) = x_1$ $f_2(\vec{x}) = g(\vec{x})\left(1 - \sqrt{\frac{x_1}{g(\vec{x})}}\right)$ $g(\vec{x}) = 1 + 10(n-1) + \sum_{i=2}^{n} x_i^2 - 10\cos(4\pi x_i)$	$x_1 \in [0, 1]$ $x_i = 0$ $(i \neq 1)$
ZDT6	10	$x_i \in [0, 1]$	$f_1(\vec{x}) = x_1$ $f_2(\vec{x}) = g(\vec{x})\left(1 - \left(\frac{f_1(\vec{x})}{g(\vec{x})}\right)^2\right)$ $g(\vec{x}) = 1 + 9\left(\frac{\sum_{i=2}^{n} x_i}{n-1}\right)^{0.25}$	$x_1 \in [0, 1]$ $x_i = 0$ $(i \neq 1)$

Table C.1: A summary of the unit tests on NSGA-II.

Appendix D

Project Proposal

Petar Veličković
Trinity College
pv273

Project Proposal

Computer Science Tripos, Part II

Molecular multiplex network inference

23 October 2014

Project Originators:
Dr Pietro Liò
Petar Veličković

Project Supervisor:
Dr Pietro Liò

Director of Studies:
Dr Arthur C. Norman

Project Overseers:
Prof Peter Robinson
Dr Robert N. Watson

Introduction and Description of the Work

Machine learning, and statistical analysis in general, are crucial methods in the development of the field of diagnostic medicine. Determining whether or not, for example, a patient is infected with a certain type of disease given the symptoms he is experiencing or laboratory results, can be ideally presented as a problem solved by a classifier. As another example, we may be interested in knowing whether the patient is at significant risk of developing a disease, or, if he/she already is affected, the likely prognosis of the development of the disease – these may often be represented as regression problems.

The importance of such models is greater than ever, now that various kinds of biomolecular data can be extracted with greater certainty. However, most current machine learning model implementations will tend to operate on a single type of data only. This can still provide us with precise inferences, however in reality most of the data types, particularly at the molecular level, exhibit a level of *correlation* that is differently pronounced depending on the biological process/disease in question. It is therefore to be expected that taking into account several data types at once and modelling their interactions correctly within our machine learning model should provide us with even better inferences. These interactions are not fully understood, but are assumed to be more complicated than what simply combining two separate structures in a predictable way can model. As such, there is a need for creating a data structure which may be trained to learn from data sets of each individual type as well as to model their correlation.

With this project I propose a possible solution to this problem – combining multiple *hidden Markov models* (HMMs) over identical sets of nodes, each of which has been individually trained on a single type of data, with additional interlayer links – this kind of multilayered graph is known as a *multiplex network*. To the best of my knowledge, there currently exists no open-source implementation of a structure like this. As multiple types of correlated data arise in a multitude of fields, it is expected that a generic implementation of this project will prove useful not only to bioinformaticians, but essentially anyone having to perform any kind of statistical analysis. The language of choice for the implementation of this project is C++, because I already have substantial experience in it, and it is optimised for performance – hence, it can accommodate evaluation on larger data sets compared to the other options I had.

Starting Point

The project implementation will draw material from the following courses of the Computer Science Tripos:

∼ **Bioinformatics** Being the central project area, this course provides an overview of biological contexts behind the machine learning models utilised, as well as examples of hidden Markov model usage in analysing biomolecular data;

$\sim$ **Artificial Intelligence I/II** The material covered in these courses is closely related to the methods utilised in this project, such as machine learning and optimisation. In particular, the standard algorithms on hidden Markov models are covered within it.

$\sim$ **Programming in C/C++** As previously mentioned, C++ will be the language of choice for the implementation of this project. I already have an extensive experience with the language, having used it primarily for competitive algorithmic programming. This course has provided me with an introduction to templates, which I will be using extensively to create as generic library elements as possible.

The Bioinformatics course is ongoing as this proposal is being written, while I will need to familiarise myself with the relevant Artificial Intelligence II material in advance. Material relevant to other aspects of the project (primarily in the form of academic papers and open-source projects) will be investigated in the initial stages, as outlined in the timetable.

Substance and Structure of the Project

As outlined in the introductory section, this project intends to produce a data structure for supervised machine learning (separate training and test data sets for purposes of classification and/or regression) which will accommodate for multiple types of data that are correlated in an undefined way with respect to the problem at hand. The key three building blocks of this data structure are as follows:

- **Hidden Markov models** (HMMs) will be used to model the solution to the problem by utilising each individual data type provided. More precisely, the full data structure will consist of multiple HMMs over the same set of nodes (for example, these may represent genes or patients), each of which has been trained on a particular type of data.

- **Multiplex networks** are then used to intertwine these individual layers together. In the most general form, a multiplex network over L layers and n nodes can be represented as an $L \times L$ matrix of $n \times n$ matrices

$$\mathbf{M} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \cdots & \mathbf{A}_{1L} \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \cdots & \mathbf{A}_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{L1} & \mathbf{A}_{L2} & \cdots & \mathbf{A}_{LL} \end{pmatrix}$$

  where $\mathbf{A}_{ii}$ corresponds to the edge weight matrix representing the ith individual layer, that is, $(\mathbf{A}_{ii})_{jk}$ represents the weight of the edge between nodes j and k in the ith layer. Similarly, $\mathbf{A}_{xy}$ $(x \neq y)$ corresponds to the interlayer connections between layers x and y. More precisely, $(\mathbf{A}_{xy})_{ij}$ represents the weight of the edge between node i in layer x, and node j in layer y. In practice, the interlayer connections are usually modelled in the form $\mathbf{A}_{xy} = \omega_{xy}\mathbf{I}$ (where $\mathbf{I}$ is the identity matrix), i.e. a

node is only linked to its own image in the other layer, and all edges between a particular pair of layers have equal weight.

A multiplex network is a special case of *multilayered graphs*, enforcing that all the layers are built over the same set of nodes. The solution to the problem of combining correlated data sets for machine learning may be pursued by considering construction of general multilayered graphs, however these fall out of the scope of the types of data used for evaluating this project.

- **Multiobjective genetic algorithms** are to be used in the final step of the construction of the data structure, which is training the multiplex – that is, determining the interlayer connection matrices $\mathbf{A}_{xy}$ (usually just the coefficients ω_{xy}, as discussed above) that will yield optimal inferences on the test set. These algorithms solve the *multiobjective optimisation problem*, which may be formulated as minimising several functions $f_1(\vec{x}), f_2(\vec{x}), \ldots, f_n(\vec{x})$ simultaneously. For the purposes of training the multiplex network, we would want to optimise the probabilities of each training set being properly classified by the model.

The execution of the project has been split into five main objectives. Their completion has been specifically marked within the milestones given in the project timetable below. The objectives are as follows, in chronological order:

- **Preparation** – this objective consists of preparatory background reading on the theory behind the data structures and algorithms given above. To confirm successful completion of this chapter as well as commence work on the full project, a working implementation of a hidden Markov model should be produced at the end of this phase and tested on data provided in the literature.

- **Core project implementation** – this objective consists of implementing the main building blocks and integrating them in a full data structure with a classification inference algorithm. Most, if not all, tests will be executed with a two-layered multiplex, but the design should be in principle extendable to an arbitrary amount of layers.

- **Core project evaluation** – this objective consists of evaluating the performance of the implementation mentioned above, both on small and large data sets. The evaluation will be performed in two main ways:

  - **Supervised learning setup**: dividing the given data sets into *training* and *testing* subsets, constructing the implemented data structure using the training set and afterwards examining the quality of its predictions on the testing set, against the known classifications provided therewith;
  - **Comparison with individual layers**: testing the gain we have made by combining different types of data. This involves constructing the individual HMM layers trained on the same data as the entire structure, using only one

of the data types provided within the data sets, and then comparing the accuracy of those single-layer classifiers with the accuracy of the implemented data structure on the testing set.

The classification problem that will be addressed in the evaluation of this project is classifying patients for diabetes. The layers of the multiplex will be trained and tested on two related types of biomolecular data that have been correlated with the presence of diabetes:

- *DNA methylation* (amount of CH_3 (methyl-) groups attached to the base nucleotides in particular genes);
- *Gene expression* (measure of the activity of transcription of particular genes into proteins).

- **Extensions** – this objective consists of augmenting the data structure with two additional functionalities identified as extensions: one of them is the addition of *regression* inference algorithms (producing continuous output as opposed to the discrete output of the classifier), and the other is introduction of tunable *"random noise"* in the structure (in the form of adding/removing nodes/edges, or modifying edge weights by a random amount), to accommodate for the slightly stochastic nature of these processes. Both extensions should be implemented and evaluated, time permitting, upon completion of the core project.

- **Dissertation** – this objective consists of writing up the project dissertation, documenting how all of the above objectives have been achieved in a clear and logical order.

Success Criteria

The project will be considered a success upon satisfactory completion of the **Preparation**, **Core project implementation**, **Core project evaluation** and **Dissertation** objectives as outlined above. Precisely:

- The complete proposed classifier data structure should be implemented, incorporating at least the three main building blocks outlined in the previous section. Correct operation of the individual modules within the structure should be tested on the sample tests provided in relevant literature or academic papers.

- The accuracy of the classifier should be evaluated with at least the two methods given in the previous section (supervised learning setup to estimate accuracy, and comparison with single-layered classifiers). It might prove useful to provide further ways of evaluation, e.g. of the structure's running time/space efficiency on the training and testing sets; these methods should be investigated as needed.

- Finally, a dissertation should be produced, documenting the necessary introduction to the problem area, the work done in the stages of preparation, implementation

and evaluation of the project, and the conclusions drawn from the project's overall execution.

The **Extensions** objective should also be achievable, however it is orthogonal to the core project and as such is not essential to its success.

Resources Required

I intend to use my own laptop (2.6 GHz Intel Core i7 with 8 GB RAM, running Mac OS X 10.10 Yosemite) for the purposes of implementation, evaluation and dissertation writeup. I accept full responsibility for this machine and I have made the following contingency plans to protect myself against hardware/software failures:

- Revision control of the project's codebase and dissertation using `git`, with all commits pushed onto a remote private repository on GitHub (with intents of making it public upon completion of the project);
- Synchronisation of the entire project with my personal file spaces on Dropbox, Google Drive and the MCS utilising `rsync`, both manually and automatically;
- Regular backups of the machine's entire filesystem (and hence the project files as well) utilising Apple's Time Machine, on an external 1TB HDD.

For evaluation purposes, the project will also utilise molecular data (DNA methylation and gene expression matrices of patients, with provided classifications for diabetes) located in the Gene Expression Omnibus (http://www.ncbi.nlm.nih.gov/geo/). This data is readily available, and I am familiar with the methods of accessing the relevant sets.

This project is not expected to rely on any libraries for the three major modules outlined in the project substance, as some proposed features such as adding random noise to the model and making the model generic may not be easily integrated with the current implementations available, justifying an implementation from scratch. However, in the event of a major schedule overrun, the project implementation may fall back to extending an existing library.

Timetable and Milestones

To facilitate execution of the project in accordance with the proposed structure, I have divided the project development timetable into fifteen fortnightly slots, some of which have been reserved for initial research and dissertation writeup, and some serving as buffer time in the event of schedule overruns. To keep track of the progress of the project, I have assigned a milestone to each slot – it should be verified at the end of each slot that its respective milestone has been achieved.

The rough plan is to complete the core project by the end of December, implement proposed extensions by mid-February, and produce a final dissertation writeup ready for submission by late April. The full proposed timetable is given below:

Slot 0: *6 October – 22 October*

→ Working on the project proposal.

→ Setup of all contingency schemes as described in the resource declaration section.

→ Studying the essentials of hidden Markov models from relevant literature and the previous year's Artificial Intelligence II course notes.

Milestone: Project proposal ready for submission.

Slot 1: *23 October – 5 November*
Deadline: Project proposal submission (24 October)

→ Researching academic papers relevant to multiplex networks, and investigating genetic algorithm implementations and their relative merits.

→ Starting work on a generic HMM implementation in C++.

Milestone: A basic working HMM implementation, tested on a few examples given in literature (**Preparation completed**).

Slot 2: *6 November – 19 November*

→ Implementation of a generic multiplex network class.

→ Integration of the HMM as a layer in the multiplex.

Milestone: Successfully connecting two HMMs in a multiplex network. Providing a front-end command line tool which can read and analyse a multiplex given in a prescribed format.

Slot 3: *20 November – 3 December*

→ Implementation of inference algorithms on multiplex networks; verification that the algorithms produce expected outputs on known networks.

Milestone: Inferences of comparable quality successfully made on known examples of multiplex networks (from research papers).

Slot 4: *4 December – 17 December*

→ Implementation of a suitable genetic algorithm to be used for training the interlayer edge weights for the multiplex network.

Milestone: Working generic implementation of the genetic algorithm, tested on example functions given in relevant academic papers.

Slot 5: *18 December – 31 December*

→ Integration of the genetic algorithm with what was previously done; utilising it to train the underlying multiplex as described.

→ Attempting a first evaluation run on a small data set; fixing any residual bugs in the design as found.

Milestone: Classifier successfully ran on a small data set (**Core project implementation completed**).

Slot 6: *1 January – 14 January*

→ Evaluating the core project by training and testing on large data sets; in particular, performing a comparison with HMM classifiers over a single data type.

→ Writing the progress report; if necessary, sending it to the supervisor and incorporating any comments.

Milestone: Progress report ready for submission. Classifier performance evaluated on large data sets (**Core project evaluation completed**).

Slot 7: *15 January – 28 January*

→ Implementing a regression method of inference on the multiplex HMM structure. Running on the provided data sets, and commenting on the obtained results.

→ Prepare slides for the progress report presentation (utilising output from the evaluation as needed).

Milestone: Successful regression performed on the multiplex, producing useful output for the given training and testing data sets. Presentation slides prepared.

Slot 8: *29 January – 11 February*
Deadline: Progress report submission (30 January)

→ Rehearse and deliver presentation to overseeing group.

→ Implementation of a feature to allow introducing random noise into the multiplex structure, either by adding extra nodes/edges or modifying the values of existing ones. Commenting on performance with respect to the noise parameter(s).

Milestone: Presentation successfully delivered. A working scheme for adding noise into the model, with appropriate testing conducted (**Extensions completed**).

Slot 9: *12 February – 25 February*

→ Buffer slot #1, to accommodate any schedule overruns with the implementation; if there are none, start work on the dissertation early.

Milestone: None.

Slot 10: *26 February – 11 March*

→ Starting work on the dissertation; completion of preparatory chapters.

Milestone: Completion of the Introduction and Preparation chapters of the dissertation.

Slot 11: *12 March – 25 March*

→ Completion of draft dissertation.

→ Submission of the draft to the supervisor and Director of Studies.

Milestone: Completion of the Implementation, Evaluation and Conclusion chapters of the dissertation. Submission of the first draft.

Slot 12: *26 March – 8 April*

→ Receipt of responses from the supervisor, Director of Studies and any proofreaders.

→ Incorporation of the obtained comments as appropriate, in preparation of the second draft of the dissertation.

Milestone: Submission of the second draft of the dissertation to the supervisor and Director of Studies.

Slot 13: *9 April – 22 April*

→ Receipt of responses on the second draft.

→ Incorporate any further comments as appropriate.

Milestone: Final version of the dissertation ready for submission.

Slot 14: *23 April – 6 May*

→ Print, bind and submit the dissertation.

Milestone: Submission of dissertation (**Dissertation completed**).

Slot 15: *7 May – 13 May*

→ Buffer slot #2, to accommodate any unexpected issues with the writeup of the dissertation.

→ Revision for Part II examinations.

Deadline: Dissertation submission (15 May)