Comparative Analysis of Route Prediction Algorithms For Service Placement in Multi-Access Edge Computing

Sherwin Mascarenhas, B.Tech.

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science (Intelligent Systems)

Supervisor: Siobhán Clarke

August 2021

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Sherwin Mascarenhas

September 7, 2021

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Sherwin Mascarenhas

September 7, 2021

Comparative Analysis of Route Prediction Algorithms For Service Placement in Multi-Access Edge Computing

Sherwin Mascarenhas, Master of Science in Computer Science

University of Dublin, Trinity College, 2021

Supervisor: Siobhán Clarke

While smartphones and other mobile devices have gone through enormous innovation in recent years, they are still resource constraint due to their size and require to offload some of the computationally expensive tasks to Cloud Computing Servers. However, Cloud Servers are plagued by low latency and bandwidth, as they are generally located remotely, in relation to the user. Multi-Access Edge Computing (MEC) is a next generation Cloud Computing architecture that brings some of the power of a Centralised Cloud Server to the edge of the radio access network. As MEC servers are deployed at the edge of the radio access network they are capable of providing application service with low latency and high bandwidth as compared to traditional Centralized Cloud Servers. MEC servers are however also resource constraint when compared to a Centralized Cloud Server, and are capable of running only a finite number of services on a single edge server. Hence, large applications have to be decomposed and distributed onto different edge servers, using a dedicated service placement algorithm that optimizes the placement of services on different edge nodes.

However, MEC faces a critical challenge, due to the increasing mobility of users. This is because, placing of services at edge servers closer the current location of the user, can be made redundant by, due to the movement of the user. This leads to increased latency and results in a decrease in the Quality of Service. Hence, service placement algorithms need to consider the future movement of the user, before placing services on edge servers.

To address these challenges, two state of the art route prediction algorithms, that are based on Sequence to Sequence and Hidden Markov Models, are implemented in this dissertation. These algorithms are then integrated into an existing Service Placement

Algorithm, to prioritize the placement of services at edge servers closer to the predicted trajectory of the user.

These route prediction models are evaluated on the accuracy of their prediction as well as the average distance of the prediction from the actual route. The performance of the route prediction algorithms are also compared, in the context of the service placement algorithm, by measuring the average network latency of the service, the average waiting time and the resource utilization of the edge servers.

The results of this study, confirm that route prediction algorithms do indeed improve the performance of service placement algorithms. Furthermore, the route prediction algorithms implemented in this dissertation outperform a baseline Cluster Based Hidden Markov Model, implemented in existing research.

Acknowledgments

I would like start off, by thanking my Academic Supervisor, Professor Siobhán Clarke for the tremendous support she has given me throughout the course of my dissertation. I really appreciate her constant feedback and motivation in helping me produce the best results.

I would also like to express my sincerest gratitude to Dr Christian Cabrera and Dr. Sergej Svorobej. They personally, took time out of their busy schedule to help guide and resolve any issues I was facing. I would have found my dissertation quite challenging to complete without them.

Finally, I would also like to give a huge shout out to my Mum and Dad and all my friends for their continuous support.

Sherwin Mascarenhas

University of Dublin, Trinity College
August 2021

Contents

Abstract **iii**

Acknowledgments **v**

Chapter 1 Introduction **1**
1.1 Motivation 2
1.2 Research Objectives 3
1.3 Outline of the Dissertation 4

Chapter 2 State of the Art **5**
2.1 Background Review 5
2.1.1 Multi-Access Edge Networks 5
2.1.2 MEC Service Placement 6
2.1.3 Mobility Aware Service Placement 8
2.2 Route Prediction 9
2.2.1 Markov Models 9
2.2.2 Recurrent Neural Networks 13
2.3 Previous Work 20
2.3.1 MEC System Architecture 20
2.3.2 Service Placement Algorithm (Ant Colony Optimization) 21
2.4 Summary 24

Chapter 3 Design **26**
3.1 Problem Overview 26
3.2 System Overview 27
3.3 Grid Based Map Matching 28
3.4 Route Prediction 29
3.4.1 Recurrent Neural Network Model 29
3.4.2 Hidden Markov Model 33

3.4.3 Cluster Based Hidden Markov Model . . . 38
3.5 Summary . . . 40

Chapter 4 Implementation **41**
4.1 Environment Setup . . . 41
4.1.1 Simulation Environment . . . 42
4.1.2 Route Prediction Environment . . . 42
4.2 Dataset and Pre-processing . . . 43
4.2.1 Taxi Trajectory Dataset . . . 43
4.2.2 Map Data . . . 44
4.2.3 Pre-processing . . . 45
4.2.4 Map Matching . . . 47
4.3 Route Prediction . . . 50
4.3.1 Sequence to Sequence Models . . . 50
4.3.2 Hidden Markov Model . . . 56
4.4 Summary . . . 59

Chapter 5 Evaluation **60**
5.1 Objectives . . . 60
5.2 Route Prediction Experiments . . . 60
5.2.1 Sequence to Sequence Algorithms . . . 62
5.2.2 Hidden Markov Model . . . 67
5.2.3 Cluster Based Hidden Markov Model . . . 69
5.2.4 Route Prediction Discussion . . . 70
5.3 Service Placement Experiments . . . 72
5.3.1 Experiment Setup . . . 72
5.3.2 Evaluation Metrics . . . 73
5.3.3 Service Placement Results and Discussion . . . 74
5.4 Summary . . . 77

Chapter 6 Conclusions & Future Work **78**
6.1 Summary . . . 78
6.2 Contributions . . . 78
6.3 Limitations . . . 79
6.4 Future Work . . . 80

Bibliography **81**

Appendices **86**
.1 Code for Route Prediction Algorithms 87
.1.1 Attention Based Sequence to Sequence Network 87
.1.2 Hidden Markov Model 90

List of Tables

4.1 Library Versions 43
4.2 Description of the inputs to the Encoder Class 52
4.3 Description of the Layers in the Encoder Class 52
4.4 Description of the inputs to the Decoder Class 53
4.5 Description of the Layers in the Decoder Class 53

5.1 Variables of the experiment 63
5.2 Variables of the experiment 72
5.3 Variables of the experiment 73

List of Figures

2.1 Structure of a Simple Markov Chain 10
2.2 Structure of a Hidden Markov Model 11
2.3 Recurrent Neural Network Architecture 14
2.4 RNN Cell Architecture 15
2.5 LSTM Cell Architecture 16
2.6 GRU Cell Architecture 17
2.7 Major Components of the MEC System 20
2.8 Ant Colony Optimization Algorithm: High Level Overview 22

3.1 Mobility Aware Service Placement Problem Diagram 26
3.2 High Level System Design 27
3.3 Grid Map with 9 Unique Grid Cells 28
3.4 Sequence to Sequence RNN Architecture 29
3.5 Hidden Markov Model: Conditional Probabilities Calculation 31
3.6 Hidden Markov Model: Conditional Probabilities Calculation 34
3.7 Example Conditional Probability Matrices for Viterbi Algorithm 36
3.8 Viterbi Algorithm 36
3.9 Cluster Hidden Markov Model Design 38
3.10 Example Cluster HMM Grouped Routes 39

4.1 Environment Setup 41
4.2 Taxi Dataset HeatMap 44
4.3 Dataset Pre-processing: High Level Overview 45
4.4 Extracted Taxi Heat Map 46
4.5 UML Diagram: Sequence to Sequence Model 50
4.6 UML Diagram: Bahdanau Attention based Sequence to Sequence Model . 55

5.1 Haversine Distances for 100 Grid Cells 64
5.2 Validation Loss for 100 grid cells 64
5.3 Average Validation and Haversine Distance 100 Grid Cells 64

5.4 Haversine Distances for 225 Grid Cells 64
5.5 Validation Loss for 225 grid cells 64
5.6 Average Validation and Haversine Distance 225 Grid Cells 64
5.7 Haversine Distances for 400 Grid Cells 65
5.8 Validation Loss for 400 grid cells 65
5.9 Average Validation and Haversine Distance 400 Grid Cells 65
5.10 Haversine Distances (Different Grid Cells) 66
5.11 Validation Loss for Different Grid Cells 66
5.12 Accuracy for different Prefix Cells 66
5.13 Haversine Distance for different Prefix Cells 66
5.14 Average Scores for Different Grid Cells 66
5.15 Haversine Distances for N number of Cells Predicted 68
5.16 Accuracy for N number of Cells Predicted 68
5.17 Average Scores for N number of Cells Predicted 68
5.18 Haversine Distances for N number of Prefix Cells 68
5.19 Accuracy for N number of Prefix Cells 68
5.20 Average Scores for N number of Prefix Cells 68
5.21 Haversine Distances for N number of Prefix Cells 70
5.22 Accuracy for N number of Prefix Cells 70
5.23 Average Scores for N number of Prefix Cells 70
5.24 Haversine Distances for N number of Prefix Cells 71
5.25 Accuracy for N number of Prefix Cells 71
5.26 Average Scores for the Route Prediction Algorithms 71
5.27 Average Latency Per Taxi Trajectory Test 75
5.28 Accuracy Utilization Per Taxi Trajectory Test 75
5.29 Average Service Time Per Taxi Trajectory Test 75
5.30 Average Latency Per Algorithm 75
5.31 Average Utilization Per Algorithm 75
5.32 Average Service Time Per Algorithm 75

Chapter 1

Introduction

There has been a massive increase in the number of mobile users over the last decade. According to Statistica.com, by 2025, the number of mobile users will grow to around 7.48 billion [31]. This adoption trend can be noticed in even backwards and remote areas. Furthermore, the popularity of mobile devices has led to a proliferation of a wide array of mobile-based services and applications. These services range from applications in e-commerce, entertainment, business and even healthcare. Moreover, the portability associated with Mobile Devices is seen as an advantage, leading to this increase in mobile-based services.

While technical innovation will continuously improve the capabilities of mobile devices, they are resource-limited due to their size restriction. Due to this, mobile devices will always be inferior in their performance compared to static devices. Moreover, mobile devices are known to suffer from poor connectivity as well as experience variable performance and reliability [41]. Furthermore, resource heavy applications like image processing, machine learning and augmented reality, as well as location-based services with heavy reliance on GPS, are gaining popularity at a rapid pace and are slowly becoming an integral part of our daily lives [15]. Due to this, mobile devices will have to rely on external resources to provide services to users while maintaining good user experience [15].

The emerging field of Cloud Computing can address some of these limitations of mobile devices by offloading any additional computational and storage requirements from the mobile device itself onto the cloud. Cloud computing provides software and hardware as a service over the internet. By doing so, mobile devices can now serve highly demanding applications on low resource devices, independent of the location, as long as there is a stable internet connection. Mobile devices access cloud-based services by routing requests through the closest base station to the Cloud Server to serve the corresponding request [23]. Due to this, Cloud-Based services can sufficiently serve computationally intensive

requests that do not have low latency requirements.

While computation offloading to cloud servers has its own merits, the excessive amounts of latency lost due to round trips, between the mobile device and the cloud, can make it impossible to rely on cloud-based services for time critical applications [23]. The growth of IoT Based applications in various fields ranging from process automation to traffic control, rely on external services to provide great precision and low latency [43]. Additionally, there is also a significant rise in the number of mobile users who are dependent on the internet for their daily needs, leading to tremendous load and an increase in congestion on the core network's capacity. (Ericsson, 2016) claims that by 2022 the percentage of mobile subscriptions that are internet-enabled will be around 90% [14]. Due to this applications that have high latency and bandwidth constraints will have to look for alternative solutions to provide the required level of performance.

1.1 Motivation

Multi-Access Edge Computing is a new paradigm in cloud computing that aims to address some of these challenges by bringing a small fraction of the computational processing power and storage of a Centralized Cloud Server closer to the user [40]. By moving servers closer to the user, the latency of the user's request can be improved by processing and serving requests right at the edge of the network, saving crucial network time and reducing the load on the network. The reduction in latency and the improvement in bandwidth also provides an opportunity for mobile devices to serve real-time and highly critical applications [16]. Furthermore, 5G, the next generation of mobile networks, along with Multi-Access Edge Technology, can drastically improve the Quality of Service for the users.

Since MEC servers are limited in their resources, a single edge server may fail to serve demanding applications and hence such application services need to be distributed between different available edge servers. To achieve this, algorithms for service placement were introduced that are used to algorithmically, divide the application's service between different edge servers and are also tasked with placing the service in edge servers closest to the user. This helps in reducing the latency while still serving larger applications. Traditionally, to serve an application request, the service request for a particular application is sent from the mobile device to the Cloud Server running the placement algorithm. The placement algorithm then fetches information about the available edge servers, like latency and available resources. Once the placement algorithm gets all the required information, it places the request among the top k servers that meet the given requirements of the application. Furthermore, the criticality of the application is also another metric that

is considered while serving application requests [8]. In this manner, traditional placement algorithms can appropriately serve application requests from mobile devices that meet their specific requirements.

Although Service Placement Algorithms can sufficiently solve most of the challenges of centralised cloud computing, MEC is facing a new challenge concerning the increasing mobility of the user [22][32]. This is because the user movement can lead to an increase in latency and delay in the application's service. It can also affect the connectivity and the experience of the user. Maintaining continuity of service and experience in the presence of user movement is quite challenging due to the inherent unpredictability of user movement [32].This problem becomes more pronounced when users rely on MEC servers for highly time-critical services where the Quality of Service of the request needs to be maintained throughout the user's journey. Furthermore, with the increasing number of IoT devices and mobile users in Smart cities, there is a need to develop a novel placement algorithm that can consider the user's Mobility. This can be done by optimising the service placement algorithm depending on the future path of the user.

To solve these challenges, a Mobility Aware Service Placement Algorithm and Route Prediction algorithm was implemented in [8], to improve the Quality of Service for mobile users. The mobility aware service placement algorithm showed an improvement in performance, among the different metrics used, against baseline service placement algorithms that were not optimized to handle user mobility. However, this system was tested on mobility traces acquired through simulation and custom routing algorithms and made use of simple clustering algorithm for route prediction. Furthermore the route prediction algorithm was not evaluated thoroughly and the viability of its performance is unknown.

Due to the lack of concrete testing present in existing work, there was a natural interest in studying different route prediction algorithms individually and in the context of service placement models as well as evaluating their performance on real trajectory data.

1.2 Research Objectives

This dissertation is based on the following research question: Can route prediction models improve the performance of service placement algorithms in Multi-Access Edge Systems under real trajectory data?

The main objectives of this study are:

- Study and implement different state of the art route prediction algorithms.

- Evaluate and Compare the individual performance of the route prediction algorithms.
- Integrate the route prediction models studied in this dissertation with a service placement model from existing work.
- Conduct extensive simulations and evaluate the performance of the service placement algorithms on real trajectory data.

To achieve these objectives in this study, a Hidden Markov Model and a Sequence to Sequence network, based route prediction models have been studied and implemented in this dissertation. These models have been integrated with a Ant Colony Based Service (ACO) Placement Algorithm, studied in [8]. Furthermore, to study whether the route prediction algorithms are capable of predicting the future trajectory of a user, they have been tested and evaluated against a baseline Cluster Based Hidden Markov Model implemented in [8]. Similarly the performance of the Mobility Aware ACO Service Placement algorithm will be compared with a general ACO based Service Placement Algorithm. The results of this study, will generate appreciable insights in the domain of Mobility Aware Service Placement Algorithms and will help shape future research in this area.

1.3 Outline of the Dissertation

The rest of this dissertation is structured as follows: Chapter 2 presents the state of the art in Service Placement and Route Prediction algorithms. It also describes the design of the MEC system and service placement algorithm implemented in existing works. Chapter 3 presents the design of the algorithms implemented in this study. Chapter 4 gives a detailed description of the implementation of these route prediction algorithms and the dataset pre-processing steps as well as introduces the simulation environment, where the service placement experiments are run. Chapter 5 discusses the evaluation process used and the results acquired. Chapter 6 Concludes the dissertation and describes the limitations and future work of this study.

Chapter 2

State of the Art

This section will present a background overview on MEC Systems and Service Placement Algorithms. This chapter will also explore the related literature on Route Prediction Algorithms and will describe some of the fundamental concepts on Recurrent Neural Networks and Markov Models.

2.1 Background Review

2.1.1 Multi-Access Edge Networks

Since the beginning, mobile devices, that include devices like smartphones, IoT based smart objects etc., were envisioned as the future of computing [34]. Many complex applications were built to be run on mobile devices which brought applications that were run on static devices like computers on mobile devices. However, since it was not possible to bring apps, that required resources that were only available on computers with large resources, to mobile devices, due to their limited storage and computing capacity, the concept of Mobile Cloud Computing (MCC) was thought up to solve the limitations of these portable mobile devices [48]. Mobile Cloud Computing was capable of moving such complex applications away from mobile devices and onto the Cloud. Using these techniques, MCC's were also able to reduce the energy consumption of mobile devices [44]. Nevertheless, the use of MCC's lead to high latencies and scalability problems and was associated with privacy and security concerns [11].

To solve the challenges associated with MCC's, the concept of Edge Computing was proposed, with Cloudlets being the first of such an architecture introduced by [42]. Cloudlets is an Edge Computing architecture that brought the computing power of Cloud servers to the edge of the network, where the placements of such servers were strategically decided. However, cloudlets were capable of supporting either Wifi or cellular networks

at a time and faced major problems with long-range connectivity [2] [33]. Similarly, Fog Computing, another edge computing architecture, was introduced by Cisco in 2012, that brought the computational power from the core of the network to the network edge [5]. Fog Computing has had widespread applications in smart cities from sensor data to connected vehicles [33]. It also helps reduce the load on the network by serving compute as well as storage solutions in proximity to the user [48]. However, for real-time services with low latency and high bandwidth demands, both these systems are incapable of providing the required quality of service.

Compared to the architectures discussed, Multi-Access Edge Computing (MEC), brings resources of a centralized cloud server from the network core to the edge of the Radio Access Network (RAN) [34]. The placement of servers close to the users, allows application services to achieve real-time latency as well as bandwidth requirements [48]. This provided a means of serving complex applications that require real-time service on mobile devices, improving the Quality of Service of the users. Compared to MCC, MEC systems are also capable of serving contextual information based on the location of the request. This opens up enormous opportunities for serving applications in the connected vehicles space as well as V2X technologies. Furthermore, localized computationally intensive tasks, like object detection and surveillance benefit, from the ultra-low latency and high bandwidth of MEC systems, as they can now avoid the sending of huge amounts of data over the network to a centralized cloud [39]. The advantages of having ultra-low latency also facilitates the use of applications such as traffic control and real-time analytics computed on sensor data collected in smart cities. By processing applications with huge data requirements at the edge, MEC's can also serve to reduce network stress on the cloud server [44]. Additionally, as MEC servers allow for flexible deployment options in areas closer to the RAN, bigger edge aggregation centres can be built to tackle the growth of traffic incurred due to the rise of IoT devices [34]. Due to this, even the growth of IoT devices as well as the collaboration of such devices in environments like, smart grids, intelligent transportation systems, and smart cities, can be easily managed in MEC systems [34].

2.1.2 MEC Service Placement

While MEC's have proven to improve the quality of service for many different applications, they are resource constraint, when compared to Cloud Servers. When coupled with an increasing number of applications and mobile network traffic, the latency, as well as waiting times for application service, can increase drastically, leading to poor Quality Of Service [48]. To tackle this challenge, the application service need to be distributed

between multiple MEC servers [48]. However, to optimize the performance of the system, the placement of services at the different edge servers has to be examined carefully [4]. Furthermore, placement of services in edge services that are already serving multiple requests can lead to multiple disruptions in the application's service, caused due to frequent disconnections [4].

Since resources on MEC servers are limited, Service placement algorithms are used to optimize the placement of service on multiple different edge servers, to improve the performance of the MEC network. These algorithms are NP-hard as they depend on multiple factors like a large number of edge servers and the number of services offered by MEC systems [59] [51]. For instance, traffic information as well as other location-based services, that are offered to a taxi need to be placed at edge servers that are located closest to the taxi. As such services require low latency and high bandwidth, MEC systems need to support not only such requirements but also multiple such requests for services that may arrive for instance, from multiple taxis.

Several different types of service placement solutions have been proposed in previous studies. Most of these solutions can be broadly classified into Exact, Heuristic and Meta-Heuristic algorithms [8].

Exact algorithms are a group of algorithms that always find optimal solutions by searching through the entire solution's space [51]. [56] uses an algorithm based on the popular exact Integer Linear Programming algorithm for service placement to optimize the cost involved with placing services at different edge servers. [57] uses an enumeration algorithm that uses dynamic programming to divide the problem into sub-problems. Compared to [56], [57] tries to optimize the traffic of data that arrives at each edge service. While such exact algorithms are capable of finding an optimal solution, they are computationally expensive as they have to explore the entire solution space before arriving at a solution [51].

On the other hand, heuristic algorithms are capable of finding optimal solutions by restricting the solution space according to a given constraint. [20] uses a Cluster-based approach for service placement along with a Greedy Algorithm to calculate the distances between Radio Access Networks and Edge servers. The paper aims to optimize the placement of services in edge servers that satisfy a certain heuristic, which in this paper is the amount of acceptable delay. Similarly, [29] uses a greedy based heuristic algorithm for service placement in V2X. The study tries to optimize the latency or delay at different edge servers. While heuristic algorithms are capable of computing solutions to problems quicker than Exact algorithms, they face a high risk of getting stuck in local maximas [8].

Metaheuristic algorithms are heuristic-based approaches, that try to find optimal solutions using an iterative process [8]. [59] uses a metaheuristic algorithm based on the

Ant Colony Optimization algorithm for scheduling services on cloud servers. This model follows a multi-objective approach where the heuristics used for optimizing the performance of the MEC system are, the time to completion of service and the cost of the service. Similarly, another study uses a model-free Duelling Deep Q Network algorithm, where a multi-objective problem is modelled as a Markov Decision Process [55]. In this study, the authors try to optimize the scheduling of services on different edge servers, based on the patterns of the user requests, the available resources on the edge servers and the data dependencies for each of the requests. Compared to Heuristic algorithms, a Meta-Heuristic based approach utilizes an iterative process, that make such algorithms capable of avoiding the local maxima problems that Heuristic algorithms face.

2.1.3 Mobility Aware Service Placement

The studies discussed above try to improve the performance of the MEC system, but do not account for the mobility of the user. The mobility of the user poses a real challenge, for MEC systems, especially for the provision of real time services that have hard constraints on the latency and delay of the service. To address this challenge, recent studies have incorporated user mobility into service placement algorithms [58], [52] [26].

[58] uses a Multi-Arm Bandit Model for service placement in an MEC system in the presence of user mobility. The model collects mobility information from the users and then uses such information as context for the service placement algorithm. The authors, make two assumptions, which are that the users are willing to share their data and the users that are in the same location at the same time, will request similar services. However, such an assumption is not generalizable, and in a large smart cities with edge servers offering many different applications such an assumption cannot be held.

[52] uses a Meta-Heuristic algorithm that is based on the Genetic and Simulated Annealing algorithm. The algorithm treats the service placement algorithm as an optimization problem, where the response time of the MEC system has to be optimized. This algorithm does show better response times when compared to other widely used heuristic algorithms the Genetic algorithm and the Simulated Annealing algorithm. However, when compared to previous works, their system relies on prior knowledge of the users mobility to build their Mobility Aware Service placement algorithm on. This study does not rely does not rely on mobility prediction systems in conjunction with a service placement algorithm to evaluate their model on real trajectory data.

On the other hand [26] uses prior user mobility information for service provisioning in MEC systems. They predict the next location of the users movement and use that to set up application services in areas with higher user requests. The heuristic algorithm

used in this paper aims to maximize the utility of the network and improve the network delay of real time applications. This paper make use of a simple auto regressive mobility prediction algorithm, however, no information is provided on the data or the performance of the prediction algorithm.

While all these studies utilize the movement of a user to place services, none have employed a dedicated route prediction algorithm, that is capable of predicting either partial or the full route of the user. In later sections we will describe how this study uses a Service Placement Algorithm that utilizes route prediction algorithms to achieve better placement performance.

2.2 Route Prediction

Route prediction algorithms have been recently gaining traction due to their many different applications. These algorithms have been commonly used for modelling location-based information such as the gathering of traffic information as well as the calculation of alternative routes that depend on current traffic information. Furthermore, these prediction algorithms have also been applied for fetching information on different points of interest close to the predicted route of the user [2].

Map Matching, is a commonly used pre-processing step, that maps GPS coordinates to unique ids, instead of using raw GPS coordinates for prediction. The more popular Map Matching technique, which is followed by [46], [10], [19], [37], maps GPS coordinates to unique identifiers of road segments. That is, GPS coordinates that fall near a particular road segment are all matched to the id of that road segment.

Since a Hidden Markov Model and variations of Recurrent Neural Network-based Sequence to Sequence models have been implemented for route prediction in this dissertation, the next section will focus on some of the algorithms that have been implemented in the literature in these respective fields.

2.2.1 Markov Models

Markov Models are probabilistic models that are based on the Markovian formalism, that in a sequence of events, future states only depend on the current states [17] . Markov Models can be divided into Fixed Order and Variable Order Markov Models.

Fixed Order Markov Models

Fixed Order Markov Models, depend on a fixed number of previous states to calculate the probability of moving to a new state. A first-order Markov Chain is the most simplest form

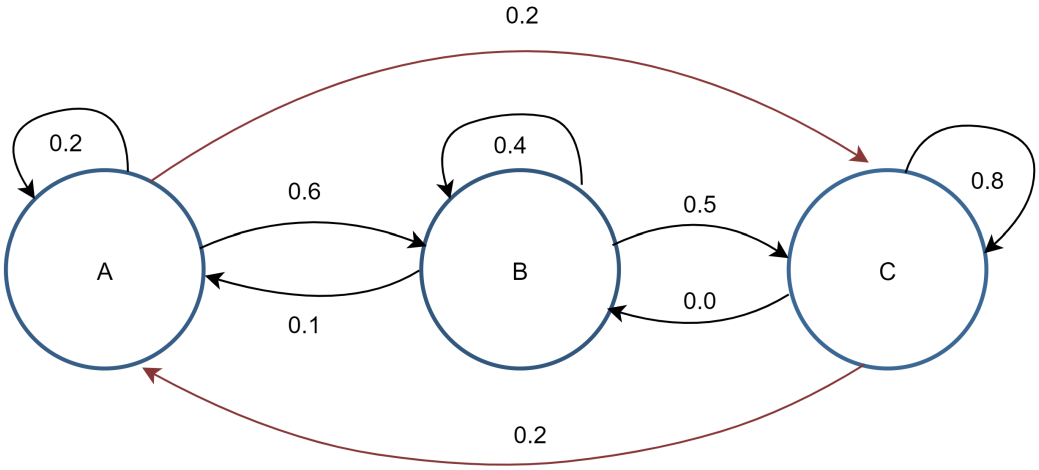

Figure 2.1: Structure of a Simple Markov Chain

of a fixed order Markov Model, where the current state only depends on the previous state. At the heart of any fixed order Markov Model, is a Probability Matrix called the Transition Probability Matrix (A) [49]. This Matrix is used to compute the probability of moving from one state to another. Equation 2.1, shows how the probability of transitioning from current state t given all the previous states, is calculated using just the conditional probability of the current state given the previous state, if the order of the model is 1. The transition probability matrix, contains all such transitions from previous states to next state.

$$P(S_t|S_1, S_2, S_3, \ldots, S_{t-1}) = P(S_t|S_{t-1}) \tag{2.1}$$

A common approach applied to calculating transition matrices are based on computing the frequencies of observations of states in the context of a given state. This approach is only viable when the data on the population is given.

Variations to Markov Chain Models, include orders of dependence on previous states greater that one. For instance, an order 2 Markov Chain are modelled using equation 2.2:

$$P(S_t|S_1, S_2, S_3, \ldots, S_{t-1}) = P(S_t|(S_{t-1}, S_{t-2})) \tag{2.2}$$

While all the states in a Markov Chain Model are observable, i.e. we can observe the values of such states, a Hidden Markov Model is another fixed order Markov Model, that can be represented by a set of Hidden States that describe the Observable States in the model. Hidden Markov Models are capable of capturing complexities in the data, that are otherwise governed by factors unknown, which is not possible in standard Markov Chain Models. Moreover, the transition probabilities in a Hidden Markov model represent the probabilities of moving from one hidden state to another. Observable states do not

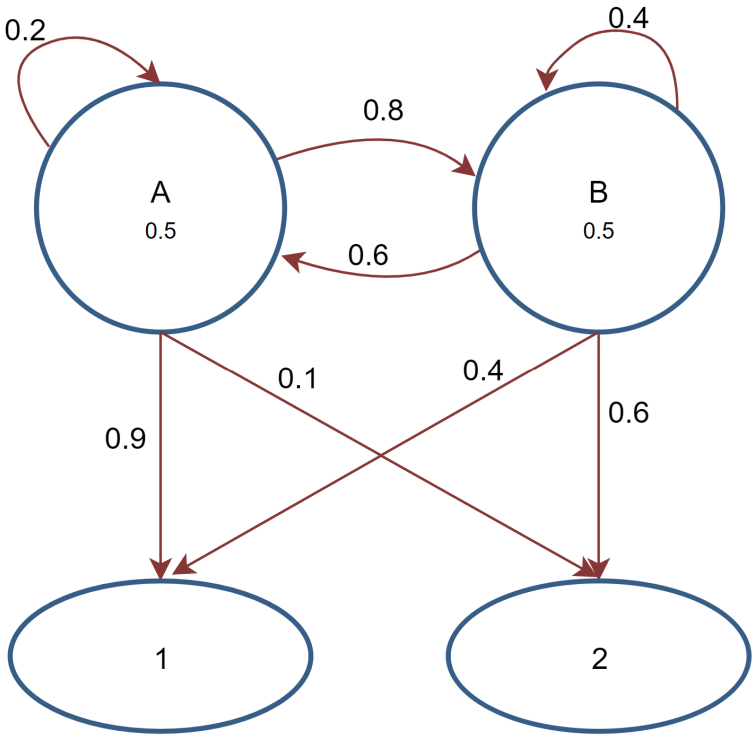

Figure 2.2: Structure of a Hidden Markov Model

have transition probabilities, however, Hidden Markov Models introduce the concept of a Emission Probability, which is the probability of observing an Observable state O_i in a particular Hidden State H_j. Equation 2.3, shows how the emission probabilities are calculated using given frequencies of the hidden and observable states.

$$P(O_i|H_j) = \frac{Frequency\ of\ O_i\ given\ H_j}{\sum_{k=1}^{N}(Freq\ of\ O_k\ given\ H_j)} \tag{2.3}$$

where, N is the number of Observable states in the model.

Variable Order Markov Models

Variable Order Markov Models are a group of Markov Models where the order of the dependence on previous states is not fixed. In contrast to fixed-order Markov Models, Variable Order Markov models learn the conditional distributions of a model based on variable contexts, which is dependent on the available statistics from the input data [49]. These models present the flexibility of modelling complex sequences, by using higher as well as lower-order dependencies [49].

2.2.1.1 Markov Based Route Prediction Algorithms

The popularity of Markov Models has grown tremendously since their roots in the 1960s when it was first applied to speech-based models [36]. After periods of continuous refinement since then, these models have been successfully used in many different applications, from natural language tasks to diagnosis of different diseases as well as the prediction of different weather patterns [21] [36].

Since Markov Models were designed to model sequential data, these models have had great success with modelling user mobility, which is evidenced by the breadth of research that is dedicated to using the Markovian Process for the task of route prediction [53] [10] [37].

(Simmons et all) conducted some of the earliest research on route prediction, where a Hidden Markov Model was used to mine trip patterns from a historical dataset of users trajectory [46]. The model also used contextual data like the destination of the user and current timestamp to improve the performance of the model. While the study does claim an accuracy of around 98%, the evaluation was conducted on a dataset that included only 46 trips. Furthermore, the prediction is based on the assumption that the driver follows a routine, which can be mined for future prediction. This assumption will work well for personalized route prediction, however, may not work as well for highly variable data like taxi data.

[53] on the other hand, improves upon this work, by using a variable order HMM on taxi data to mine mobility patterns by using a Probabilistic Suffix Tree for route prediction. They used Multiple VMM's to model different mobility patterns that are influenced by different traffic conditions. The model was also evaluated on real taxi trajectory data from the city of Shanghai.

[10] proposed a personalized route prediction system that uses a first order markov model to predict the intended route of the user, which is also capable of predicting deviations in the driver's intended route. However, the model has been designed to predict the route given information about the origin and destination of the user. The model is evaluated on trips collected from two different drivers, where the model is able to predict with an accuracy of 97% for driver 1 as compared to 65 % for driver 2. Moreover, the evaluation was conducted on only a maximum of 81 trips and the poor performance of trips is because of the inability of the algorithm to optimize for larger trips.

Compared to the building of a standard Hidden Markov Model for route prediction, [19] uses a frequency co-occurrence matrix instead of transition and emission probabilities to build a simple, lightweight HMM model for the route as well as destination prediction. While the model is able to achieve an accuracy of about 94%, the dataset used to evaluate

the model was generated using simulation, with only variations of 23 origin and destination pairs being used. This approach has been used as our baseline model in this study, an will be tested an evaluated on real trajectory data.

[30] uses a Variable Order Markov Model to predict the destination as well as the route of a user. The model also is capable of predicting routes in real-time and can dynamically predict deviated routes.

[37] uses an advanced clustering algorithm with a trajectory prediction model based on Markov chains. This paper was designed specifically to scale to big data trajectories involving dense urban trajectory data.

[17] uses a traditional Hidden Markov Model for modelling the mobility of a user, where an optimized version of the Viterbi algorithm is used for prediction. Compared to the previous studies that map GPS coordinates to identities of different roads on the map, a grid-based map matching algorithm is used in this study to map GPS coordinates to the identities of grids, that divide the map. However, the model is capable of only predicting the next sequence in a route. In our study, we use an adapted version of the work done in this paper, and we modify it to predict future trajectory sequences of any length.

In our study, we chose to adopt a Hidden Markov Model as one of our route prediction models, due to its ability of modelling sequential data well. While other Markov Models like Variable Order Markov Models have shown great performance for the task of route prediction as well, they suffer due to their computational complexity that grows exponentially with the increase in the context vector. Additionally, given the number of studies that have utilized HMMs effectively for the purpose of route prediction, it felt like a reasonable choice to implement and compare such a model against other more complex models.

2.2.2 Recurrent Neural Networks

Out of the various deep learning algorithms, Recurrent Neural Networks (RNNs) have also been modelled specifically to deal with applications involving sequential data. Compared to vanilla neural networks, RNNs do not require any assumption of independence between the data points. This is desirable, as there are many such applications, where data points related in time need to be modelled. For instance, in natural language processing, sentences are related in time, which fails the independence assumption that standard neural networks require [24]. Furthermore, Neural Networks cannot deal with variable length sequences, which limit their applicability, to many different sequence based tasks [24]. Moreover, when compared to Markov Models, recurrent neural networks are capable

of working with longer sequences [24]. This is due to the incapability of fixed size Markov Models from drawing on dependencies from timesteps older than the order of the model.

RNNs are Feed Forward Neural Network, where the outputs from each RNN layer are fed as inputs to the next layer [24]. Figure 2.3, depicts the architecture of a simple RNN network unfolded in time. Each RNN layer is governed by an RNN cell. These cells come in different variations and are used for different purposes. Three main types of RNN Cells are commonly used. These are the Standard RNN Cell, the Long Short Term Memory Cell and the Gated Recurrent Unit Cell.

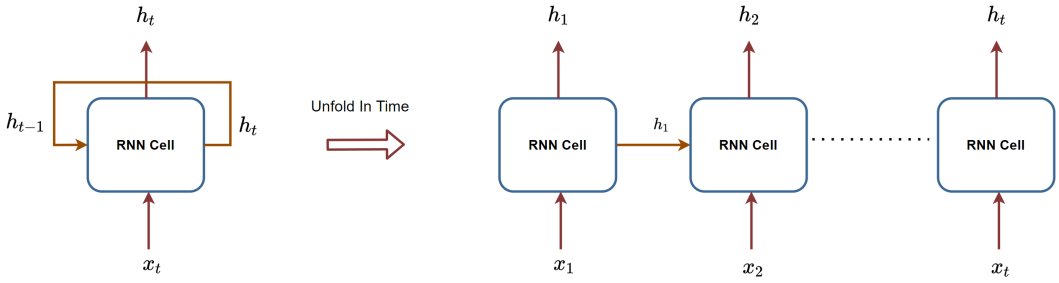

Figure 2.3: Recurrent Neural Network Architecture

Standard RNN Cell

The standard RNN cell is what is commonly refer to when RNNs are discussed in general. The standard RNN cell contains a hidden state h_t, which is a simple multi layer perceptron with an activation function. The hidden state is a vector representation of the past and current events.

$$\begin{aligned} h_t &= tanh(W_h * h_{t-1} + W_x x_t + b) \\ y_t &= softmax(h_t) \end{aligned} \tag{2.4}$$

Where h_t is the hidden state at the current timestep, h_{t-1} is the hidden state from the previous timestep, and x_t is the input for the current timestep. W_h and W_x are the trainable weight matrices and b is the bias. In the RNN cell, the activation function used, is a tanh function and the softmax function converts the h_t into probabilities, using which the predictions are made.

Long Short Term Memory (LSTM) Cell

The Standard RNN Cell suffers from capturing long term dependencies [9] [3]. This is because gradients reduce to zero or can explode when the sequence that is trying to be

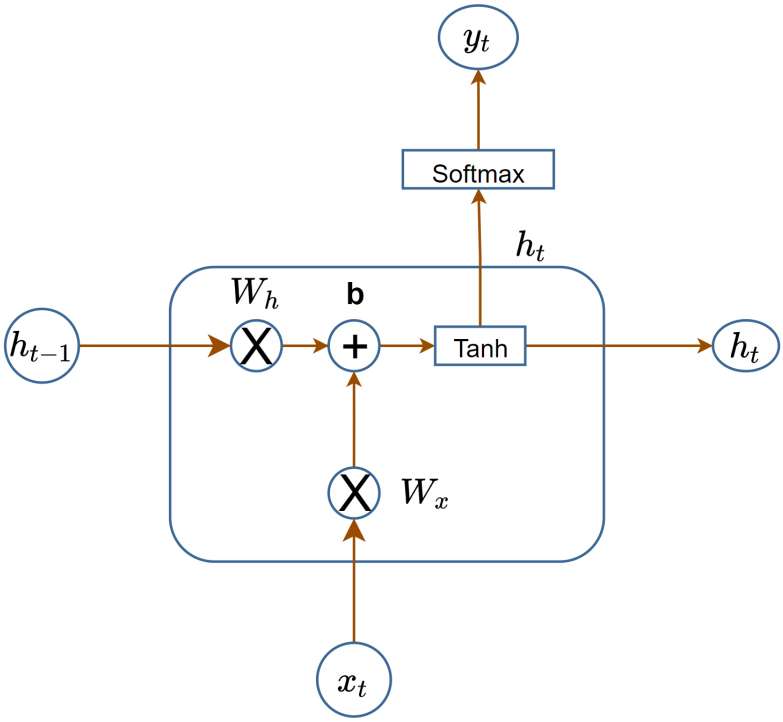

Figure 2.4: RNN Cell Architecture

modelled is very long [3]. This problem is also generally known as the vanishing gradient problem in vanilla RNNs [3]. The LSTM cell tries to solve this problem, by improving the memory capacity of the RNN cell. This is achieved with the introduction of a gating mechanism, which controls the effects of previous and current timesteps on the internal state of the cell. The forget gate, the input gate and the output gate are the three main gates that are commonly present in an LTSM cell. Different variations of the LSTM cell also may have missing or additional gates present. Apart from the various gates that are present in an LSTM cell, a memory cell, most commonly known as the cell state, is another vector, similar to the hidden state, that is present in the LSTM cell. The cell state is responsible for storing long term memory in the LSTM cell, while the hidden state is responsible for the current working memory. The cell state vector is modulated using the input gate and the forget gate, to modify the information stored in the cell state from previous and current timesteps. Due to this the LSTM layer has 2 different output vectors, that are the hidden state and the cell state.

Equation 2.5 lists the different equations for the LSTM cell [9]:

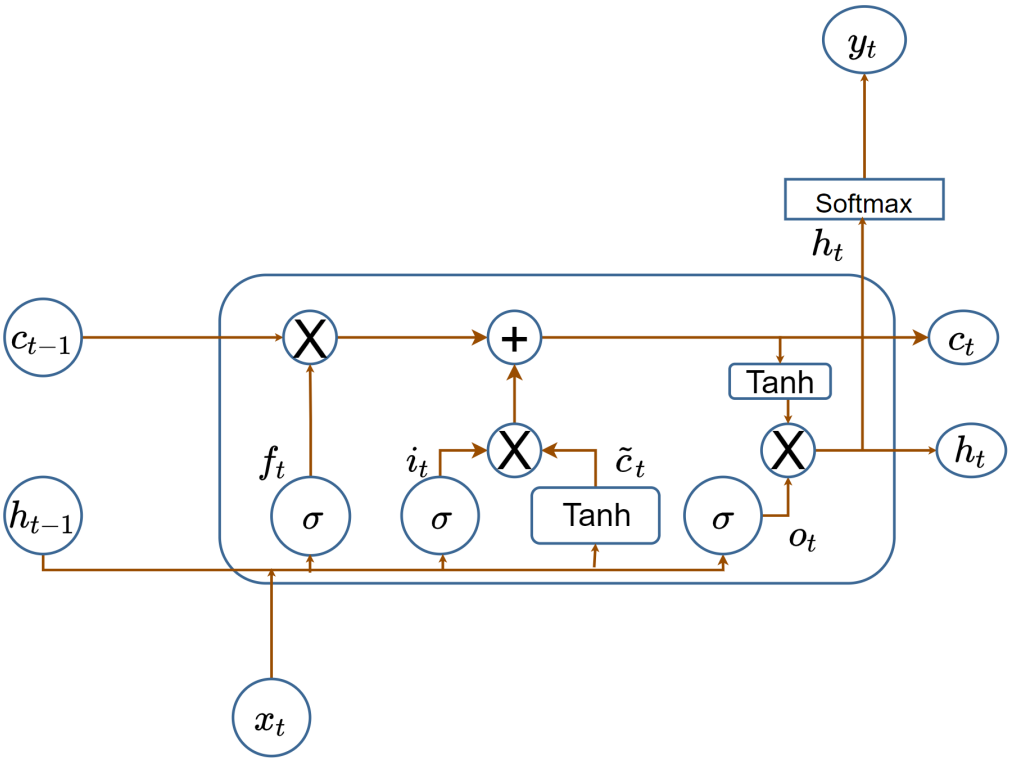

Figure 2.5: LSTM Cell Architecture

$$
\begin{aligned}
f_t &= \sigma(W_{fh}h_{t-1} + W_{fx}x_t + b_f) \\
i_t &= \sigma(W_{ih}h_{t-1} + W_{ix}x_t + b_i) \\
\tilde{c} &= tanh(W_{\tilde{c}h}h_{t-1} + W_{\tilde{c}x}x_t + b_{\tilde{c}}) \\
c_t &= f_t \cdot c_{t-1} + i_t \cdot \tilde{c}_t \\
o_t &= \sigma(W_{oh}h_{t-1} + W_{ox}x_t + b_o) \\
h_t &= o_t \cdot tanh(c_t)
\end{aligned}
\tag{2.5}
$$

In the equations given above, f_t represents the forget gate, i_t the input gate and o_t the output gate. x_t is current input data point, h_t is the hidden state, c_t is the cell state for the current LSTM layer and the $c_{t-1}^{\sim}$ is the candidate cell state vector for the current LSTM layer. Figure 2.5 shows, the architecture of the LSTM cell. The plus sign represents weighted addition and the cross sign represents the product of different intermediate values in the cell.

The Input gate governs what information is saved from the new data point at the current timestep, in the cell state, while the forget gate governs what existing data needs to be forgotten in the cell state acquired from previous timesteps. The effect of the different gate vectors on the cell state is calculated by passing these vectors through a Sigmoid

activation function, which passes the data forward if the value is 1 and forgets the data if the value is 0. Finally, the output gate controls what information from the previous hidden state and the current cell state is added to the current hidden state. Similar to the standard RNN cell, the hidden state is overwritten at every timestep, while the cell state is capable of saving information of interest from timesteps many layers deep. In this way, the LSTM cell is capable of achieving long term memory in the LSTM cell.

Gated Recurrent Unit (GRU) Cell

Major state of the art algorithm were being replaced by RNN based algorithms, because

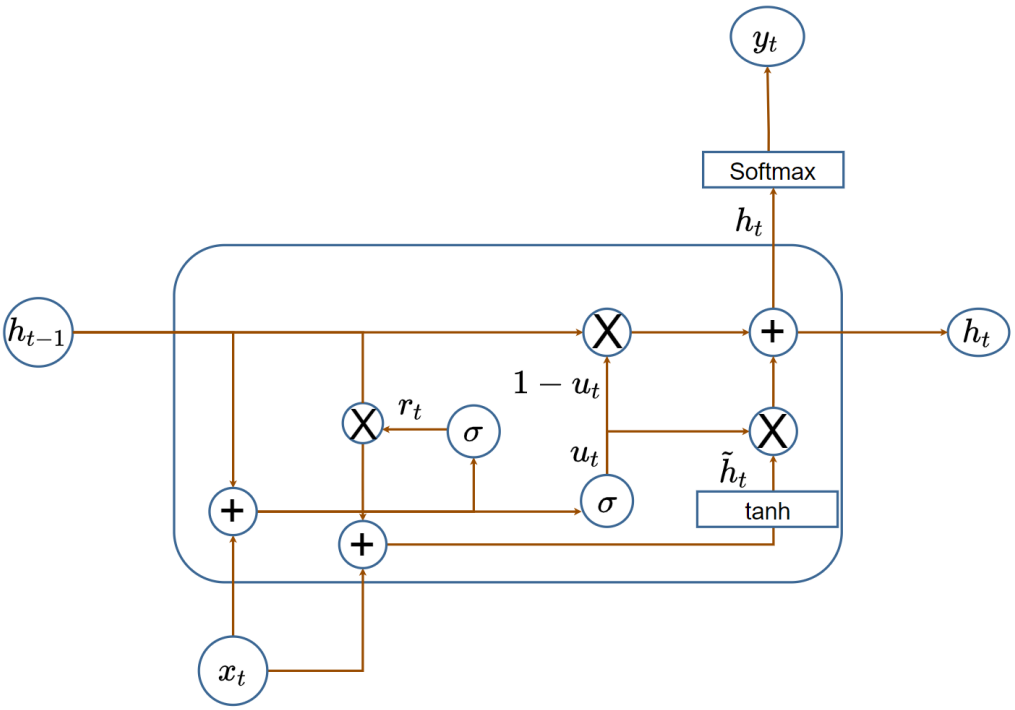

Figure 2.6: GRU Cell Architecture

of the introduction of the LSTM layer. However, the introduction of the many gating mechanism and separate memory cell, made the task of training an RNN network with LSTM cells computationally expensive. The GRU cell was introduced by [6], to try to resolve some of these challenges by reducing the complexity of the cell, while maintaining the superior performance of the LSTM cell. The GRU cell contains only 2 gates as compared to the LSTM. These are the update gate and the reset gate [54]. To reduce the complexity of the cell, the GRU layer, merges the input and forget gate of an LSTM cell into a single update cell [54]. The following are the equations for the GRU cell [54]:

$$
\begin{aligned}
r_t &= \sigma(W_{rh}h_{t-1} + W_{rx}x_t + b_r) \\
u_t &= \sigma(W_{uh}h_{t-1} + W_{ux}x_t + b_u) \\
\tilde{h}_t &= tanh(W_{\tilde{h}h}(r_t \cdot h_{t-1}) + W_{\tilde{h}x}x_t + b_u) \\
h_t &= (1 - u_t) \cdot h_{t-1} + u_t \cdot \tilde{h}t
\end{aligned}
\tag{2.6}
$$

In the equation, r_t is the reset gate, u_t is the update gate, $\tilde{h}_t$ is the candidate hidden state vector, h_t is the current hidden state. σ is the sigmoid activation function. Compared to the LSTM cell, the GRU cell, does not have a cell state.

Vector Representation

Vector representations of data are easily processed by many machine learning models as they improve the model's performance. This is because, many categorical variables in their natural form, are not easily interpreted by machine learning models, as these models primarily deal with values that have some form of relationship between them. Therefore, various techniques are used to convert raw categorical data, into machine understandable form. Due to this, Recurrent Neural Network models also only deal with vector represented input data.

The most simplest form of vectored categorical data is the one hot encoded vectors. One hot encoded vector are binary vectors with a length equal to that of the number of categorical variables in the vocabulary. These vectors are zero vectors with the exception of the index of the categorical variable that is marked as a one. For instance, if there are 5 categorical variables, that are integers from 1 to 5, then one hot encoded vector for 1 is [1,0,0,0,0].

However, recent studies have used vector embeddings for representing categorical values, instead of one hot encoded vectors, where the categorical value is transformed into a vector of dimension d. The vector embeddings are either trained separately or during the training process of the prediction algorithm, to convert data into a vector space, where they are more closely related. Embedding vectors can improve the training time of the machine learning models by using a a smaller dimension embedding as compared to one hot encoding vector, which is especially beneficial when the number of categorical values are large. However, this can also lead to a minor loss in performance. In this dissertation, we use One Hot Encoded vectors, since the performance gains, between the 2 vector representations were negligible.

Relevant Work

Since the introduction of the LSTM layer, Recurrent Neural Network models have had huge success, with applications ranging from speech recognition to language translation

to even traffic prediction [45] [27]. Due to the recent successes that RNNs have had, there have been a few studies that have applied variations of the RNN network to route prediction.

One such study implements and evaluates the performance of a Sequence to Sequence model with a LSTM Cell for mobility prediction [50]. Compared to traditional approaches, [50] argues that the use of continuous locations for prediction is able to solve the problem of inaccuracies occurring during the discretization of taxi locations which can be caused due to the presence of trajectory data with varying timesteps. However, this approach cannot be applied to datasets with some missing timestamps for each location. Furthermore, the model introduced in this paper can only predict the next location of the taxi future route (t + 1) given the previous trajectories of the user (1,2,3,...t). This again does not take full advantage of a Sequence to Sequence RNN model that allows for variable route prediction.

Compared to [50], [25] implements a doubly stacked two-layer GRU based RNN network that predicts future route segments of the taxi trajectory. It also uses a grid based map matching technique to convert the GPS positions into a sequence of grid cells identities similar to [17]. The paper compares the model's performance with an n <3 order Markov model and a particle filter algorithm, where the Deep RNN network shows better performance when evaluated with Precision, f1 score and Hamming loss. However, since the model relies on a simple RNN network instead of an Auto Encoder model, the model is incapable of working with and predicting variable-length sequences.

As compared to the previous studies, [13] uses a unique space partitioning approach based on kd-trees that divides the map according to the density of taxi routes in the dataset. The paper then proposes a simple LSTM based neural network to predict the destination of the taxi. It then calculates the top k routes of the user, using the predicted destination as the context. The problem with such an approach is the cost associated with a wrong destination prediction. Since the prediction of the destination guides the prediction of the taxi's route, the performance of the destination prediction should be reliable enough even if a small percentage of given route has been traversed. However, on evaluation, the destination prediction algorithm has a distance from the true destination, greater than 2 Kms when only 25% of the route is traversed, and greater than 1 km when 60% of the route is traversed. Such an approach if applied to retrieve the available edge servers for service placement can lead to higher latencies and frequent service placement inaccuracies due to the prediction of highly inaccurate routes during the start of the taxis trip.

From studying the literature, it is evident that only a few studies make use of variants of RNN based networks. However, these prediction models are incapable of dealing with

variable length sequences, which is crucial for effective mobility optimized service placement algorithms, as the service request from the user, may not always contain previous trajectory of length similar to what the model is capable of working with. Hence, in our study we implement and evaluate variations of Sequence to Sequence models, that have the flexibility required for predicting future route segments of variable sizes.

2.3 Previous Work

In this section, some of the important work done in [8], that this is dissertation is dependent on, will be discussed. This includes a MEC System and a Mobility Aware Service Placement Algorithm. Furthermore, a Cluster Based Hidden Markov Model from [8], used as the baseline route prediction model in this study, will also be discussed in the next chapter.

2.3.1 MEC System Architecture

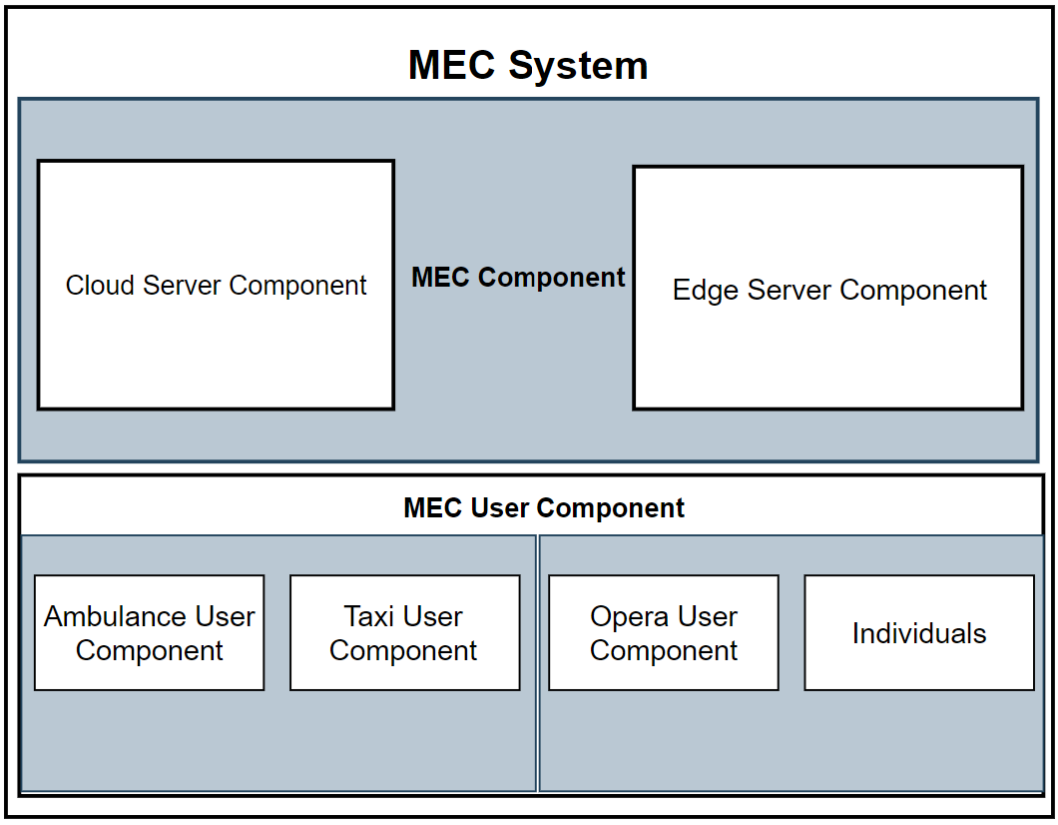

Figure 2.7: Major Components of the MEC System

Fig 2.7 shows the high level architecture of the MEC System implemented in [8]. There

are three relevant components, in the context of this dissertation, that will be described below:

- Cloud Server
  The cloud server is responsible for communicating with each and every edge server over the network using Ethernet communication protocol. It tracks the edge servers present in the system and is also responsible for scheduling and placing of services on different edge servers. The cloud server manages the Service Placement Algorithm, and in the case of Mobility Aware Service Placement, also manages the Prediction of the users' future path.

- Edge Servers
  The Edge Servers are responsible for serving user requests at the edge of the radio access network. Each server can be represented by the total resources on the server TR_i, available resources AR_i, reserved resources RR_i and the location of the edge server loc_i [8]. Resources R_i are made up of cpu ri^{cpu} and ram ri^{ram} levels. The MEC system is also used for tracking the different edge servers in the MEC system.

- MEC User Component
  The MEC user component is used to model user behavior. It can be divided into static and mobile users. Static users are users that do not change their position with time. For instance, [8] have studied the performance of placement systems in the context of opera users. Mobile users are users that are capable of changing their location with time. A mobility aware service placement algorithm was studied in the context of Ambulance users in the [8]. Since this dissertation is dealing with taxi user mobility, a new user component has been implemented that captures the mobility of the taxi users.

The high level flow of this system in reference to service placement, can be summarized as, a user sends a request for a service to the closest Edge Server. This request is then sent to the Cloud server component, where the service Placement algorithm is triggered, the Cloud Server is then responsible for physically placing the services on the selected edge servers.

2.3.2 Service Placement Algorithm (Ant Colony Optimization)

We re-use the service placement algorithm implemented in [8], to evaluate the performance of our route prediction models in the context of service placement. This placement algorithm is based on the Ant Colony Optimization algorithm, which is a meta-heuristic

swarm intelligence algorithm [12]. The intuition of the algorithm is based on techniques inspired from Ants to find the shortest path to a food source [12]. Ants use pheromones to mark pathways to food sources that are closer in distance. The more the amount of pheromones left behind, the higher the probability of the ants picking that path and the higher are the changes of picking the better path. This repetitive process finally converges to all the ants traversing on the path that was found to be the most optimal given the current task. This same technique is applied to many different fields from scheduling to routing [12].

A similar approach is used to select the servers for placing services at. Apart from the use of pheromones, to mark better placement of services, a heuristic is used as well, which is based on the amount of cpu and ram available on an edge server. Figure 2.8 shows a high level overview of the ACO based Service placement algorithm.

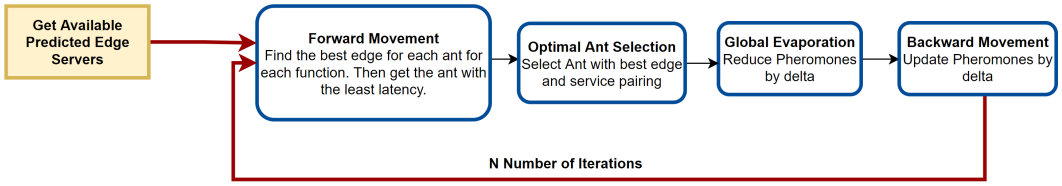

Figure 2.8: Ant Colony Optimization Algorithm: High Level Overview

This Ant colony Optimization algorithm is dependent on the following variables:

- Number of Ants, N
- Number of Services M
- Number of Edge Servers L
- Number of Iterations I
- Pheromones,P of size L X M
- Ant Matrix, A of size N
- Heuristic, H of size L X M
- EdgeProbability, prob of size L
- alpha α, influences the selection of edge servers based on available pheromones
- beta β, influences the selection of edge servers based on pheromones

- Delta δ, amount of pheromone deposit
- Placement Pl, of size M

Each ant has a memory associated with it, for every service. This is used to save the identity of the best edge server for every service in each ant. On every iteration this memory is re-written, to store the best service and edge server combination.

The ACO algorithm can be divided into four steps. The forward movement, Optimal Ant Selection, backward movement and global evaporation.

Forward Movement

Algorithm 1: Forward Movement

Given ant n and service m
for *i from 1 to L* **do**
  if *L(i) can fit* (r_i^{cpu}, r_i^{ram}) **then**
    h = $0.5 * r_i^{cpu} + 0.5 * r_i^{ram}$
    $prob(i) = P(i,n)^{\alpha} + h^{\beta}$
  end
end
j = Select random Edge with prob $\geq$ random(0,1)
ant(n) + memory(j, m)

The forward algorithm is run for every ant and for every service required by the user. For a particular ant and service, the Forward algorithm computes a probability that is dependent on the pheromone and the heuristic magnitudes. The effects of the pheromones and heuristic on this probability is controlled using the alpha and beta parameters correspondingly. An exploration step then selects a random edge with probability greater than a random number between 0 and 1. This is used as a regularization step to explore some of the other edge servers before the selecting the best one, that aids in avoiding local optimas.

Optimal Ant Selection

Algorithm 2: Forward Movement

latencies = [N]
distances = [N]
for *i from 1 to N* **do**
  latencies(i) += ant(i).memories.latencies
  distances(i) += ant(i).memories.distances
end
optimalAnt = argmin(latencies*0.5 + distances*0.5)

In the Optimal Ant selection step, the memories of the ants modified in the forward movement step are used to retrieve the ant with lowest combination of latency and distance from user.

Global Evaporation
In this step all the pheromones are reduced by δ.

Backward Movement
In this step the pheromones for the optimal ant will be increased by δ.

All these steps in order are repeated for I number of iterations. Finally, on the last iteration, from the memories of the optimal ant, the edge servers that were chosen by the algorithm is returned to the cloud server. The cloud server will then place the individual services on the difference edge servers as required.

For purposes of integrating mobility prediction, the ACO algorithm is fed only available edge servers that are closest to the future trajectory of the user. In this way the service placement algorithm can optimize its performance according to the user's mobility.

The route prediction algorithms in [8], were trying to model Ambulance mobility. However, due to the lack of Ambulance mobility data, algorithms, like Graphhoper, were resorted to, to generate traces between accident and hospital locations in the city of Dublin. Moreover, the route prediction algorithms were based on simple clustering and probabilistic methods, that were able to perform quite well, due to the lack of stochasticity in such user generated datasets, as well as the use of very small dataset for prediction. Additionally, only one or two route alternatives between these locations were added to the training dataset, which resulted in a total of only 44 training samples. These mobility models were trained on this small dataset, with no evaluation conducted on the individual performance of these algorithms.

Therefore in this dissertation we try to rectify most of these challenges, by using a Real Trajectory dataset.

2.4 Summary

In this chapter, a background overview of MEC Systems and Service Placement algorithms was given. Furthermore, various papers that implemented service placement algorithms were discussed, and the importance of mobility aware service placement algorithms was highlighted.

Furthermore, different Markov and RNN based route prediction models in the literature were discussed as well. The reasoning behind choosing a Hidden Markov Model and Sequence to Sequence RNN model was also described.

Finally, this chapter was concluded, by describing the MEC system and the Service Placement Model, implemented in an existing work, that this dissertation depends on.

Chapter 3

Design

This chapter outlines the different approaches taken to design the route prediction algorithms implemented in this dissertation. It also introduces and describes the design of the route prediction algorithm implemented in [8], which is used as a baseline model for this study.

3.1 Problem Overview

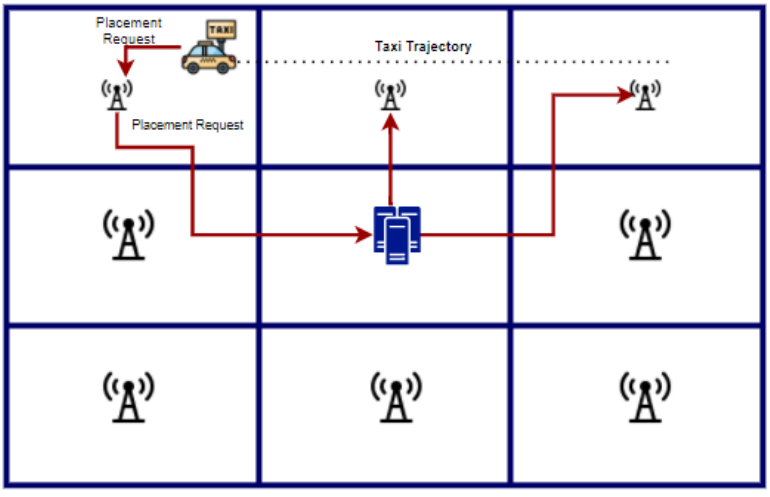

Figure 3.1: Mobility Aware Service Placement Problem Diagram

We have highlighted in previous chapters, the challenges associated with MEC systems in providing low latency and high bandwidth services to users on the move. It was discussed that most service placement algorithms place services at edge servers that are closer to the user at a point in time. Due to the mobility of users, such service placements are made redundant, as users are no longer close to the edge server that the service was

originally placed on. Hence, it is understood that service placement algorithms need to prioritize the placement of services with consideration of the future route of the user. To tackle this, our study relies on dedicated route prediction algorithms to predict the future route of the user so that services can be placed along the predicted path of the user. By maintaining low latency and high bandwidth throughout the journey of the user, the Quality of Service for the user will also improve.

In this dissertation we use a Real taxi trajectory dataset to build and evaluate our models on. Traditional route prediction algorithms assume that human trajectory contain regular or noticeable patterns[17]. While this assumption may hold true for most regular individuals, it is not true in the case of taxi user mobility. This is because taxis usually do not follow similar routes daily, as pickup and drop off points of customers are mostly random. Hence the systems implemented in this dissertation, should be able to account for the complexity of taxi mobility [17]. By building route prediction algorithms on such mobility traces that are highly stochastic in nature, these models can be re-used and applied to other mobility traces easily.

3.2 System Overview

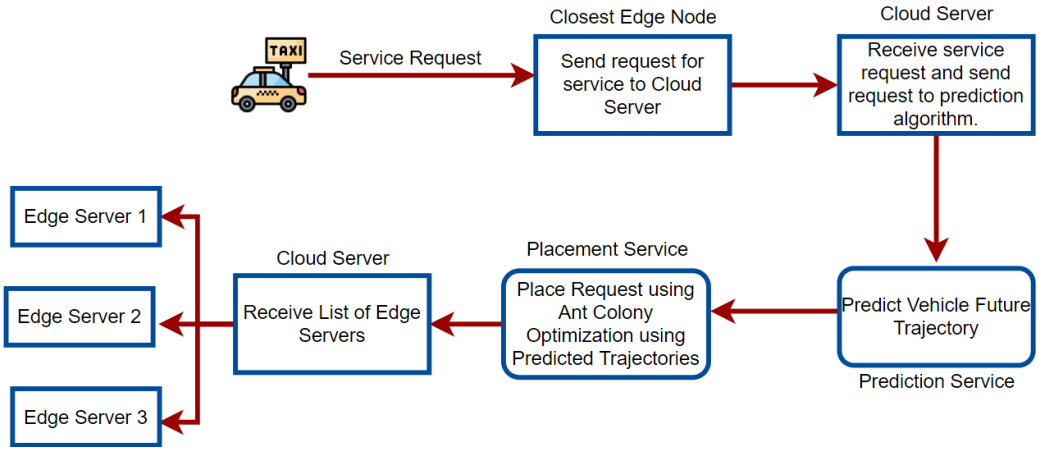

Figure 3.2: High Level System Design

This study uses a Mobility Enhanced Service placement Algorithm based on Ant Colony Optimization (ACO) and a MEC system built in [8] to evaluate our models on. We implement a Sequence 2 Sequence Encoder-Decoder model and an attention based Sequence 2 Sequence Encoder-Decoder model as the two recurrent neural network models for route prediction. Due to the popularity of Markov Models, especially Hidden Markov Models for route prediction in the literature, we also implement a Hidden Markov Model inspired from work done in [17]. Finally, we also extended a Cluster based Hidden Markov

Model implemented in [8] to compare the performance of our route prediction algorithms with.

Figure 3.2 shows a high level design of out system. In our system, Service requests from taxi users will be sent to the closest edge node. This request will then be forwarded to the Cloud Server where the route prediction algorithm is called using the trajectories of the taxi so far. Then the ACO based Service Placement Algorithm from [8] will be initiated to place services on different edge servers. Then the dedicated route prediction algorithms will predict the future trajectory of the taxi, by which available edge servers closest to that future route can be acquired. These edge servers will then be sent to the Service Placement algorithm to place services on appropriate edge servers. The Service placement algorithm will then return the list of available edge servers, on which services have to be placed by the cloud server.

3.3 Grid Based Map Matching

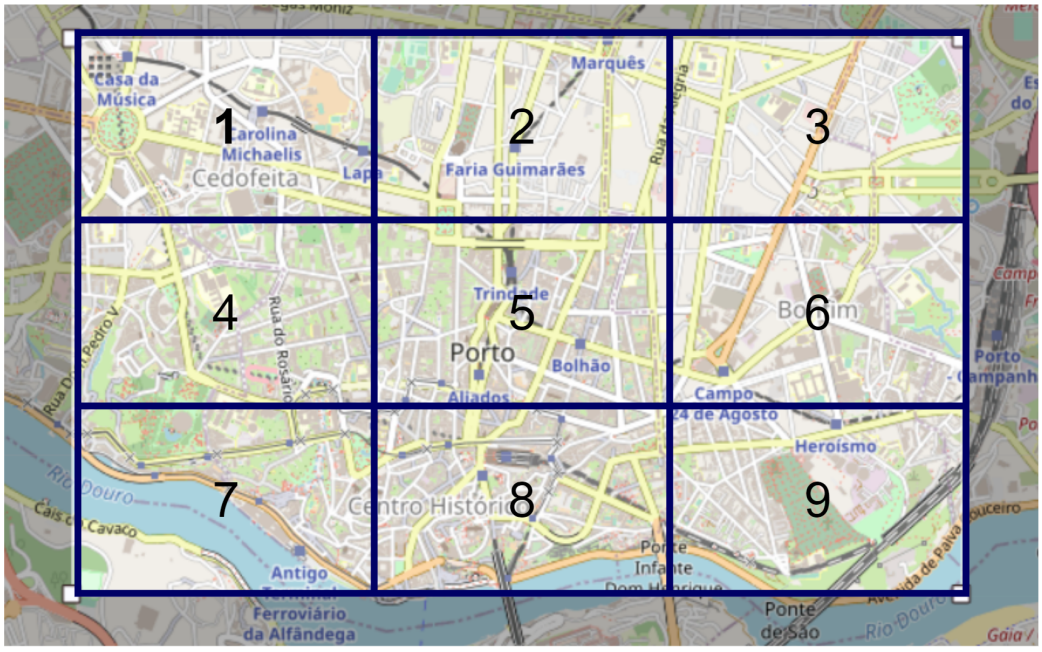

Figure 3.3: Grid Map with 9 Unique Grid Cells

To improve the computational performance of the route prediction models, these model will deal directly with discrete data. Due to this, GPS coordinates that are continuous in nature, need to be discretized. To achieve this, in this dissertation, we use a grid map representation of the original map, where the area of the map is divided into N number of cells. Each cell in the grid map is a region of space that has a unique id. Vehicle

movement in the map can now be denoted by a sequence of grid cells instead of GPS coordinates.

3.4 Route Prediction

This section will describe the design of the route prediction algorithms.

3.4.1 Recurrent Neural Network Model

Recurrent Neural Networks are well suited for the task of sequence modelling. They had great success in recent times in the domain of natural language processing. The models that have had the most success with complex sequential data are based on the concept of an encoder-decoder network. These networks are commonly referred to as Sequence to Sequence models. In our dissertation, multiple variations of the Sequence to Sequence model have been studied and implemented. The next sections will describe the design of these models in this study.

3.4.1.1 Sequence to Sequence Networks

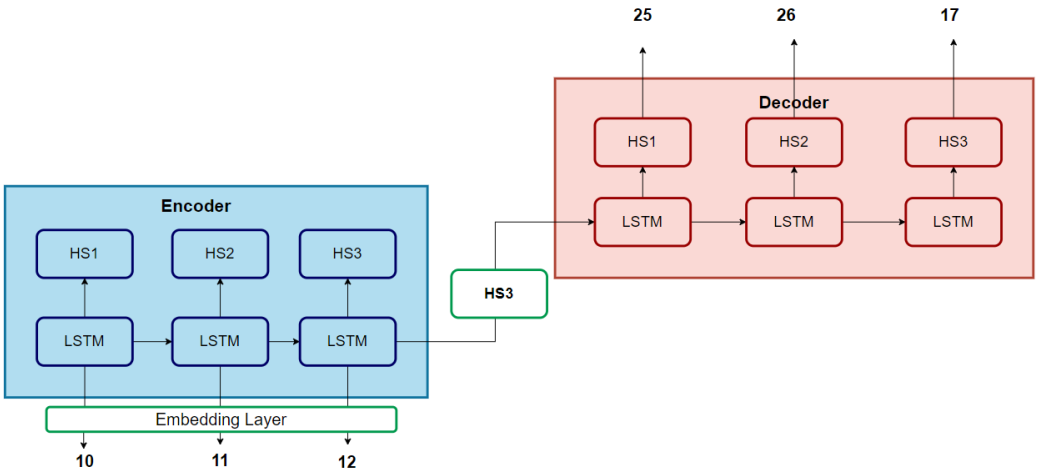

Figure 3.4: Sequence to Sequence RNN Architecture

Standard RNN networks are capable of mapping input sequence to output sequences, with good performance, whenever the length of the input and the output sequence is known ahead of time. [47]. Since, the partial trajectory of the user and the future route prediction are variable length sequences by themselves, the use of a standard RNN cell does not justify the loss of flexibility associated with it. [47] proposed a new neural network architecture that divides the input network and the output network. By dividing

the RNN into two separate networks, the length of the partial route sequence and the predicted route sequence are now independent of each other and can have size of variable lengths.

Figure 3.4 shows the architecture of the standard Sequence 2 Sequence model. The sequence to sequence model is also known as an encoder decoder model, as the first RNN network acts as the encoder, that encodes the input sequence in a fixed size vector and the decoder network, decodes the output sequence using that fixed size vector. In the case of LSTM cells, the fixed size vector contains the cell state as well as the hidden state of the last LSTM cell in the encoder network. For GRU and standard RNN cells, the fixed size vector only comprises hidden states of the last layer of the encoder network.

In the encoding network, the input sequence is read and a fixed size vector is returned. The equations for the encoder network are given as [47]:

$$\begin{aligned} h_t &= f(x_t, h_{t-1}) \\ c &= q(h_1, h_2, ..., h_T) \end{aligned} \tag{3.1}$$

Where f and q are non linear functions, h_t is the hidden states at time step t and c is the fixed size vector.

The decoder network predicts the output sequence Y, given he previous predicted data points. Equation 3.2, shows the mathematical representation of the output sequence and equation 3.3, shows how the current data point y_t at time t is predicted using the previously predicted data points, with the help of the context vector and the hidden state.

$$p(Y) = \prod_{t=1}^{T} p(y_t|(y_{t-1}, y_{t-2}, .., y_1), c) \tag{3.2}$$

$$p(y_t|(y_{t-1}, y_{t-2}, .., y_1) = a(y_{t-1}, c, hd_t) \tag{3.3}$$

Where, Y is the is the entire output sequence, a is non linear function that represents the RNN cells, hd_t is the hidden state of the decoder RNN cell at time t and c is the fixed size vector, also known as context vector from the encoder.

In the next section, we will discuss possible limitations of the standard Sequence to Sequence network and introduce a new Attention Based Sequence to Sequence model.

3.4.1.2 Attention Sequence to Sequence Model

The standard Seq2Seq Encoder-Decoder model proposed by [47], only uses a fixed size vector, i.e. the context vector, for decoding the output sequence. [1] claims that this approach, restricts the performance of the model, and that the model should be capable

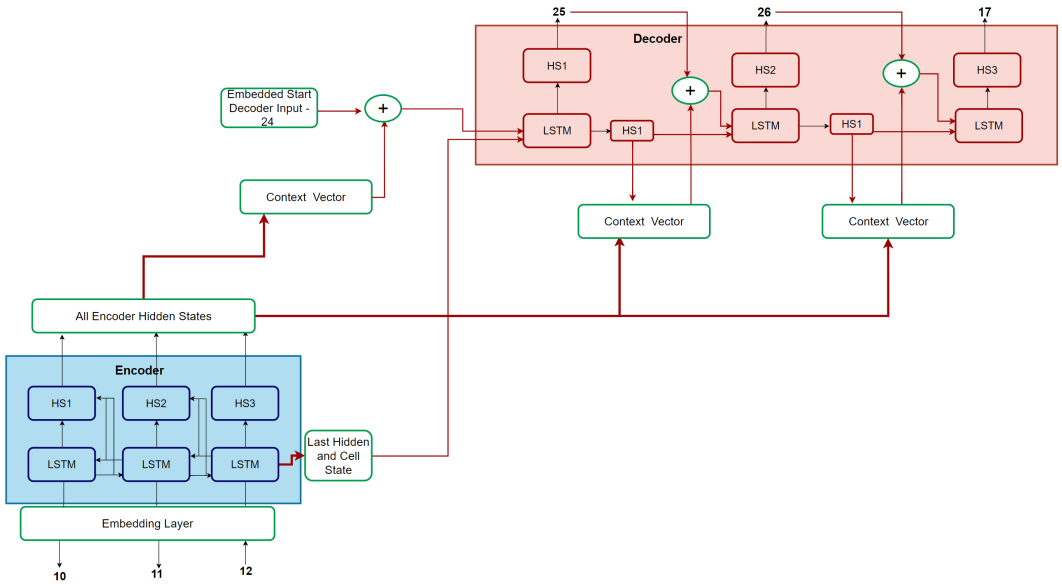

Figure 3.5: Hidden Markov Model: Conditional Probabilities Calculation

of retrieving parts of the input sequence that are relevant for the prediction of particular data point y_t in the output sequence. This approach of scanning the source sequence for relevant information to be used in the decoding process, is known as the attention mechanism in RNNs. For instance, while both the standard Seq2Seq model and the attention seq2seq model have been applied successfully, to Machine Translation tasks, the attention model shows superior performance due to the in built attention mechanism, that focuses on words in the input sequence that are highly correlated with words in the output sequence. While there are many studies that have proposed their own attention mechanism, we used the original implementation of attention mechanism studied in [1].

Figure 3.4, shows a simplified structure of the attention model. Compared to the standard Sequence to Sequence model, we can observe that all the encoder hidden states contribute to the decoding process. This is achieved by creating context vectors using the hidden states from all the encoder layers. For each prediction of a output data point at point t, a separate context vector is trained and used that pays attention to specific values in the source sequence.

The encoder network for the model can be formulated as follows [1]:

$$p(y_t|y_{t-1}, y_{t-2}, ..., y1) = g(y_{t-1}, hd_t, c_t) \tag{3.4}$$

Where, hd is the decoder hidden layer at time t and c_t is the context vector for the current timestep. Each timestep has a separate context vector as compared to [1]. g is a non linear function, that represents a RNN cell and y_t is the output from decoder at the current timestep.

$$c_t = \sum_{i=1} T_x W_{ti} he_i \tag{3.5}$$

Here, W_{ti} is the weight matrix for the current encoder layer i. T_x is the number of time steps in the input sequence or the number of encoder network layers and he_i is the encoder hidden state for layer i.

The context vector vector is dependent on the calculation of alignment scores e_{ti} and the weight matrix W_{ti}, that score Hidden States in the Encoder Network according to their importance for the current decoder output. The equation of the alignment score is given below:

$$e_{ti} = a(hd_{t-1}, he_i) \tag{3.6}$$

The alignment score for encoder hidden state i and decoder output t, is dependent on the i^{th} hidden state in the encoder and the previous hidden state of the decoder. a represents a trainable simple Multi-Layer Perceptron that generates a vector of scores for each timestep in the encoder network. This simple MLP function is also trained along with all the other components in the model. The alignment scores are then used for the calculation of the weight matrix.

$$W_{ti} = \frac{\exp(e_{ti})}{\sum_{j=1}^{T_x} \exp(e_t j)} \tag{3.7}$$

The weight matrix is calculated using a softmax of the alignment scores, where T_x is the length of the Input sequence.

From figure 3.5, it is shown that the encoder network uses a bidirectional layer. In a bidirectional layer, the hidden states are sent in both the forward and backward direction. This provides the network with context about not only the previous states, but also about the forward values in the network. A bidirectional layer is used, to allow the hidden states in the encoder network to not only summarize input sequences up and until time t but also sequences following time t. It allows the model to better represent recent data points in and around the current data point of interest in the encoder layer [1]. The encoder layer now comprises of both forward and backward hidden states.

$$\overrightarrow{He} = (\overrightarrow{He_1}, \overrightarrow{He_2}, \overrightarrow{He_3}, .., \overrightarrow{He_{Tx}}) \tag{3.8}$$

In Equation 3.10, $\overrightarrow{He}$ represents the forward hidden states, with T_x being the Number of Encoder layers.

$$\overleftarrow{He} = (\overleftarrow{He_1}, \overleftarrow{He_2}, \overleftarrow{He_3}, .., \overleftarrow{He_{Tx}}) \tag{3.9}$$

While, $\overleftarrow{He}$ represents the backward hidden states.

The hidden states for each layer in the encoder is then concatenated before being used for the generation of the context vectors.

$$\overleftrightarrow{He_i} = (\overrightarrow{He_i}, \overleftarrow{He_i}) \tag{3.10}$$

The reason a Attention Based Sequence to Sequence model has also been implemented in our work, is to study whether paying attention to different data points in the source sentence, in a route prediction algorithm, can help improve the performance of the model. These models are used generally for Natural Language processing tasks, like sentence completion and machine translation, that have high correlation between the input and output sequence data points many layers deep. For instance, in a Machine Translation task, the start of the source sentence can contain crucial information required while decoding the target sentence. However, in the domain of Route Prediction, there has been no research that studies the dependence of the target sequence on, the first few values in the input sequence. However, what we do see in the literature is that most models instead rely on only the last few inputs of the partial trajectory, which can explain the popularity of Markov Models for route prediction. During the evaluation of the sequence to sequence models, we will also highlight whether a dependence on the first few input data points exists.

3.4.2 Hidden Markov Model

The second route prediction model, that has been implemented and studied in this dissertation, is a Hidden Markov Model, inspired from work done in [17]. Compared to ordinary Markov models, Hidden Markov Models have observed states as well as hidden states. In our study, the observed states are all the unique grid cells that belong to the grid map and the hidden states are unique sub-sequences (partial-routes) of trajectories in the dataset, with a fixed size.

Given below are the definitions for the different states in the HMM:

- The observed states in the Hidden Markov model are the list of all unique cells from the grid map.

  $$O_j = C_j \tag{3.11}$$

  Where O_j is the j^{th} observed state and C_j is the j^{th} cell in the grid map. The total number of Observed States present in the model can be represented by J.

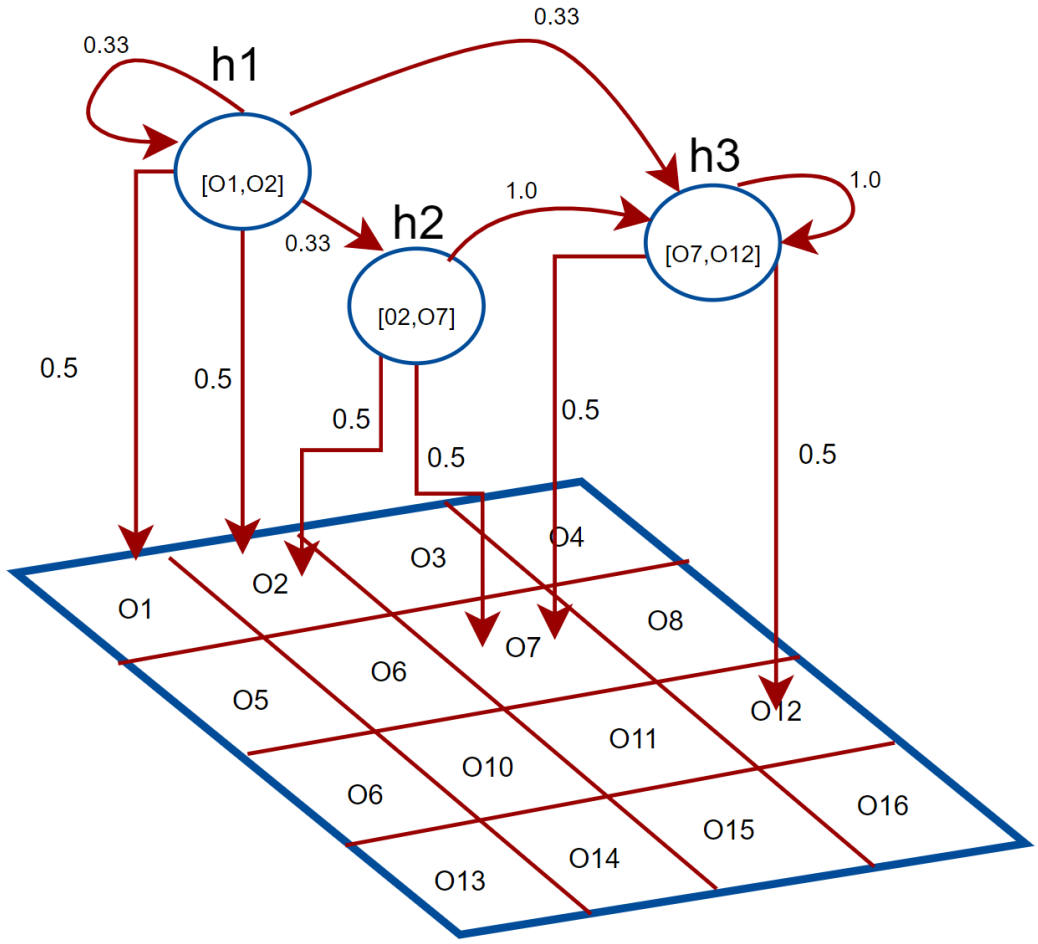

Figure 3.6: Hidden Markov Model: Conditional Probabilities Calculation

- A route in the dataset, is the list of continuous cells, traversed by a taxi.

$$R_k = [O_{j,1}, O_{j,2}, O_{j,3}, .., O_{j,n}] \tag{3.12}$$

  Where, R_k is the k^{th} unique route in the dataset.

- Hidden States are sub sequences of routes of a fixed size.

$$H_i = [O_{j,1}, O_{j,2}, O_{j,3}, .., O_{j,m}] \tag{3.13}$$

  Where, H_i is the i^{th} hidden state. Since hidden states are sub trajectories of routes, m<n. The number of Hidden States in the model is represented by N.

Statistical Properties of HMMs

A Hidden Markov Model can be built by the calculating three statistical properties of the Markov model. These are the Transition probabilities, the Emission probabilities and the Initial Probability.

- **Transition Probability**
  Hidden Markov Models follow the Markov assumption that the state at time t is dependent only on the state at the previous time step. This assumption is then applied to the hidden states of the Markov Model to calculate the probabilities of transitioning from one hidden state to another. Due to this, the calculation of the transition probabilities boils down to simple conditional probability theory. The transition probability between hidden states can be calculated as:

$$P(H_{i+1}|H_i) = \frac{\#(H_{i+1}, H_i)}{\sum_{x=1}^{N} \#(H_x, H_i)} \tag{3.14}$$

  Where # represents the number of occurrences.

- **Emission Probability**
  The probability of observing a Observed state O_j at Hidden State H_i is known as the Emission Probability. Similar to Transition Probabilities, Emission Probabilities are a conditional probability that is calculated as follows:

$$P(O_j|H_i) = \frac{\#(O_j, H_i)}{\sum_{x=1}^{J} \#(O_x, H_i)} \tag{3.15}$$

- **Initial Probability**
  The probability of starting at Hidden State H_i, is known as the Initial Probability. The initial Probability can be calculated as follows:

$$P(H_i) = \frac{\#H_i}{\sum_{x=1}^{N} \#H_x} \tag{3.16}$$

By using the formulas for each of the different probabilities mentioned, a State Transition Probability Matrix, a Emission Probability Matrix and a Initial Probability matrix can be created, which will be denoted by A,B,π.

Viterbi Algorithm

The task of retrieving the sequence of hidden variables that explains the sequence of observation, is known as the decoding problem in Hidden Markov Models. In Fig

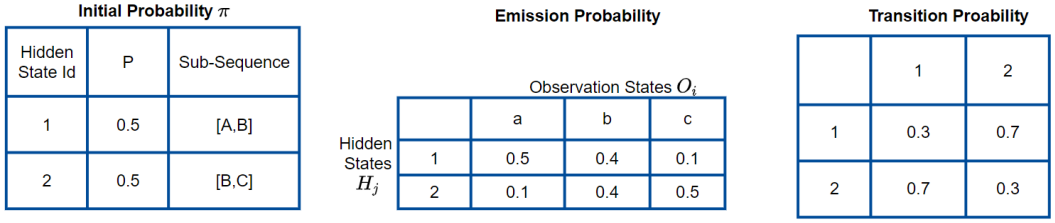

Figure 3.7: Example Conditional Probability Matrices for Viterbi Algorithm

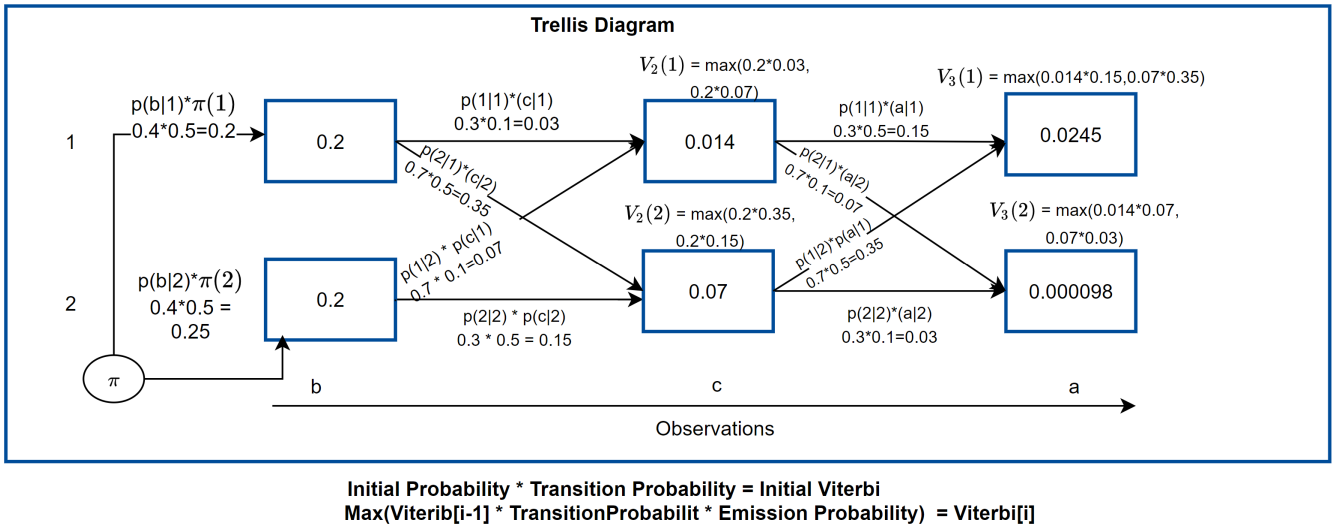

Figure 3.8: Viterbi Algorithm

3.4, example conditional probability matrices that represent the Initial Probability, the Emission Probability and the Transition Probability are given. For the given sequence of observations, b,c,a, given in Fig 3.7, the corresponding Hidden States that best explains the sequence of observed states, needs to be determined. The most common decoding algorithm used for Hidden Markov Model is the Viterbi algorithm, which is a dynamic programming algorithm, that finds the Maximum Likelihood Estimate, of hidden states given a sequence of observed states. This Viterbi Algorithm is used to predict the route of the user given the observed sequence at up until time t. In other words, given the sequences of grid cells, the most likely sequence of Sub-sequences has to calculated.

The Viterbi algorithm is based on the forward algorithm that calculates the probability of observing a sequence of observed states given a HMM model. However, compared to the forward algorithm, the Viterbi algorithm calculates the Maximum values, instead of the sum of probabilities at time t.

The viterbi algorithm uses a trellis structure for calculating the maximum probabilities, as shown in Figure 3.8, which is used in many dynamic programming algorithms [18].

The Trellis Diagram describes the Viterbi Algorithm using the following steps [18]:
Initialization

$$\begin{aligned} &N = Number\ of\ Hidden\ States \\ &T = Number\ of\ ObservationsObserved \\ &V_1(i) = \pi(H_i)B(O_1, H_i) \quad 1 < i < N \\ &Bt_1(i) = 0 \qquad\qquad\qquad\quad 1 < i < N \end{aligned} \tag{3.17}$$

Induction

$$\begin{aligned} &V_t(i) = \max_{1 \leq i \leq N} V_{t-1}(H_i)A(H_j, H_i)B(O_1, H_i) \qquad 1 < t < T1 < i < N \\ &Bt_{t-1}(i) = \operatorname*{argmax}_{1 \leq i \leq N} V_{t-1}(H_i)A(H_j, H_i)B(O_1, H_i)1 < t < T1 < i < N \end{aligned} \tag{3.18}$$

Termination

Return Bt and V matrices (3.19)

The forward pass of the Viterbi algorithm can be described by three steps. These are the Initialization, Induction and Termination steps. The Initialization step is responsible for initializing the Viterbi matrix V of size NxT and the Backtracking matrix Bt of size Nx(T-1), where N is the number of Hidden States and T is the number of partial route observations. As seen Fig 3.8, the Viterbi variable is responsible for storing the calculated intermediate values. The Viterbi value for time step t is calculated using the Transition Probability, the Emission Probability and the value of the Viterbi Matrix at time t-1. This calculation is done for each of the unique hidden states. Only the value of the maximum path for each Hidden State is stored. The back tracking variable of size Nx(T-1) stores the index of the hidden states from the previous timestep that contributed to the Maximum probability. This same process is repeated for all the time steps. After the forward pass, we calculate the Output Sequence St using the backtracking matrix.

The steps for retrieving the decoded hidden state sequence is as follows:

Initialization

$$
\begin{aligned}
&\text{T is the Number of Observations} \\
&\text{N is the Number of Hidden States} \\
&St_T = \underset{1 \le i \le N}{\operatorname{argmax}}(V_T)
\end{aligned}
\tag{3.20}
$$

Induction

$$
\begin{aligned}
&H_i = St_{t+1} \\
&St_t = Bt_t(H_i) \\
&1 \ge i > N \qquad T - 1 \ge t > 0
\end{aligned}
\tag{3.21}
$$

Termination

$$
\text{Return St} \tag{3.22}
$$

To retrieve the sequence of Hidden States, we loop backwards from timestep T to 1. On each step we find the index of the Hidden States of with the maximum probability at time t. Following these steps we are able to fetch the sequence of Hidden States that best explains the Observation sequence.

3.4.3 Cluster Based Hidden Markov Model

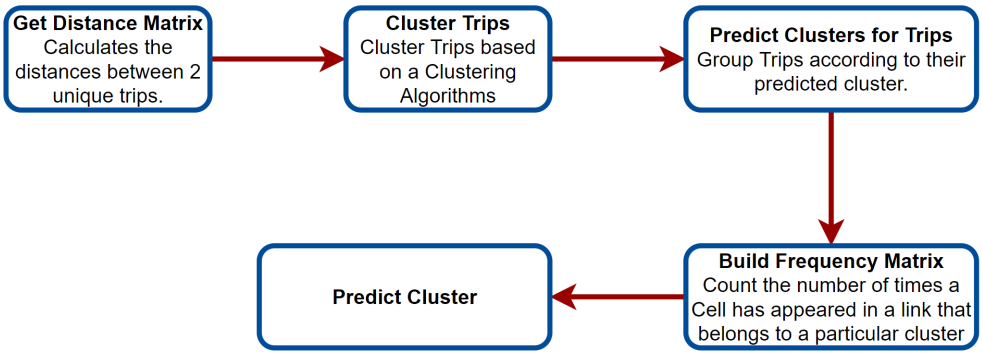

Figure 3.9: Cluster Hidden Markov Model Design

In our study we use a Cluster Based HMM, implemented in our existing work [8], as our baseline and third route prediction algorithm. This algorithm is based on the Hidden Markov Model studied in [19].

The training phase of this algorithm includes calculating distances between unique trips in the dataset, clustering trips using a clustering algorithm based on the distances calculated, and then building a frequency co-occurrence matrix F, of size mxn where m

is the number of unique cell blocks and n is the number of clusters. The frequency co-occurrence matrix, measures the frequency of being in cell c_t given that the trip belongs to cluster k_j. Using F, the following probabilities can be computed:

$$p(c_t|k_l) = \frac{F_{c_t,k_l}}{\sum_{i=1}^{m} F_{c_i,k_l}} \tag{3.23}$$

$$p(k_l|c_t) = \frac{F_{c_t,k_l}}{\sum_{j=1}^{n} F_{c_t,k_j}} \tag{3.24}$$

The prediction phase of the algorithm is based on the following recursive equation:

$$p(k_l|c_0, c_1, ..c_t) = p(c_t|k_l)p(k_l|c_0, c_1, c_2, ...c_{t-1}) \tag{3.25}$$

where, $p(k_l|c_0, c_1, c_2, ...c_{t-1})$ can be further broken up into:

$$p(k_l|c_0, c_1, ..c_{t-1}) = p(c_{t-1}|k_l)p(k_l|c_0, c_1, c_2, ...c_{t-2}) \tag{3.26}$$

$c_0, c_1, c_2, ...c_t$, are the observed trajectory of the user and k_l is the cluster that is this partial trip belongs to. This formula is based on the first order Markov process, where the current state is dependent only on the previous state.

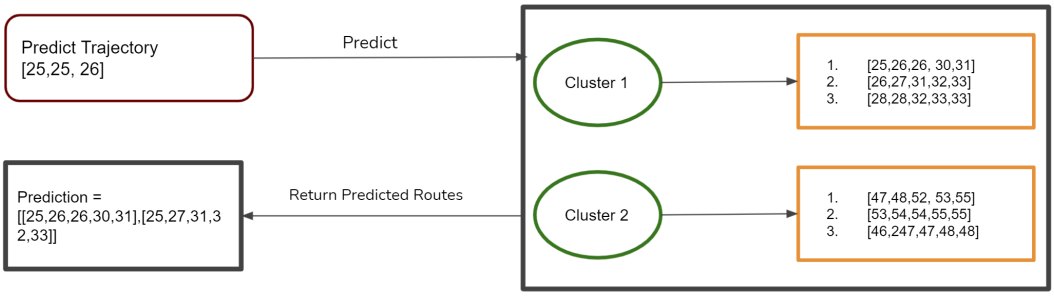

Figure 3.10: Example Cluster HMM Grouped Routes

This algorithm is called a Hidden Markov Model since the clusters here behave as hidden states and the cells are the Observable states. Figure 3.10, gives an example of how the different routes present in the dataset will be grouped. This algorithm assumes, that once a trip begins, the driver can't change his or her trip. That is they follow the same trip without the destination of the trip changing midway. Because of this, the algorithm doesn't depend on transition probabilities, as the Hidden States do not follow a sequence of changes. Instead the algorithm makes uses of a frequency co-occurrence matrix as discussed above and the prediction formula can be reduced to Naive Bayes as shown in equation 3.25

3.5 Summary

This chapter focused on the design of the algorithms implemented and studied in this dissertation. The chapter was introduced by the problem and potential solutions, and a high level overview of the design of our system was also described. Then a new map matching technique was proposed that was based on similar techniques used in the existing studies. Next the design behind the route prediction algorithms studied in this dissertation was discussed. The various sequence to sequence algorithms that were used were described and some of the advantages and disadvantages of the 2 algorithms were highlighted. The design of the Hidden Markov Model implemented in this study was described next, where the Viterbi algorithm was studied with the help of a Trellis diagram. Finally, this chapter was concluded by describing the design of the Cluster Based Hidden Markov Model implemented in an existing work, that is used as our baseline route prediction algorithm.

Chapter 4

Implementation

Until now, we have talked about the challenges of MEC systems in the face of user mobility. We have explored potential solutions and have described the design of some of the algorithms in the previous chapters. This chapter will focus on the implementation of the various techniques that were discussed previously.

4.1 Environment Setup

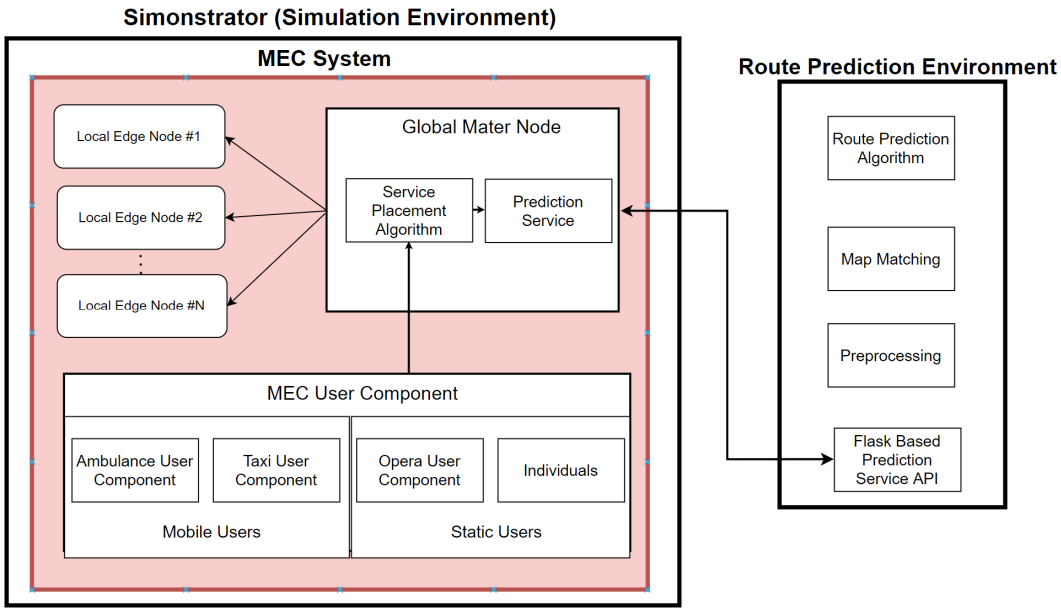

Figure 4.1: Environment Setup

Figure 4.1, shows the two different environments in which our algorithms have been implemented in. The following sections describe these environments.

4.1.1 Simulation Environment

In this paper, we use a MEC System and a Mobility Aware Service Placement Algorithm implemented in [8], to test the performance of our route prediction algorithms. The MEC System is implemented in a simulation environment to easily run experiments on the System under different configurations and parameters. Simonstrator is a simulation framework implemented in [38], which was used in [8] to build their MEC System on. It was primarily designed for Distributed Applications and has been built using the Java Programming Language. Hence the MEC System, the Service Placement Algorithm as well as the route prediction algorithms implemented in [8], have all been implemented in JAVA and integrated into Simonstrator.

In this dissertation, a prediction service as well as an additional MEC user component have been implemented into Simonstrator. The Prediction service is used to access the prediction algorithms built using Python and the, MEC User Component, also called the Taxi User Component, has been built to add custom taxi movement into Simonstrator.

4.1.2 Route Prediction Environment

The route prediction environment consists of the environment in which the Rout Prediction Algorithms, Grid Based Map Matching and other dataset prepossessing steps have been implemented. Python has been used in this environment due to its ease of use and the support the language has for machine learning research.

Some of the support frameworks that were used are:

- Numpy Library
  Numpy has been used for mathematical calculations and array processing. Numpy has been implemented in python as well as in C which speeds up the execution time of the program.

- Pandas Library
  The pandas library is used for data manipulation and has been heavily used in this study for working with csv datasets.

- scikit-learn
  scikit-learn is a popular machine learning library used in this study for creating train and tests datasets. With the help of this library the train and tests datasets, were created with fixed random shuffling, that allowed the same data to be trained and tested on each of the different prediction algorithms.

Table 4.1: Library Versions

Library Versions	
Library Name	**Version**
Pandas	1.2.4
Numpy	1.19.5
Pytorch	1.6.0+cu101
Scikit-learn	0.24.2

- Pytorch
  Pytorch is a machine learning framework, that is used to implement the standard and attention based Sequence to Sequence models. While there are other popular Machine Learning frameworks as well, pytorch has been used extensively for research prototyping, as it allows for the code to be written in a more natural pythonic syntax. Furthermore, pytorch is faster than other libraries like tensorflow, which speeds up the implementation process.

- Flask
  Flask is a light weight web server framework built for python. Flask is used in this study to enable the communication of the prediction algorithm implemented in python with the service placement algorithm written in Java.

4.2 Dataset and Pre-processing

This section will introduce the real-trajectory dataset used in this dataset, the Map data that the system is modelled using and the pre-processing steps used to process the data to be trained on the route prediction models.

4.2.1 Taxi Trajectory Dataset

The dataset used in this dissertation, is a real taxi trajectory dataset, containing trajectory data on 442 different taxis. This dataset contains taxi trajectories collected from the city of Porto, for a total of one year, between July 2013 and June 2014. The dataset was created in [28], and is freely available on the UCI Machine Learning Repository. Each taxi is fitted with a GPS recording device, which is sent and aggregated through a central taxi dispatch system [17]. Each sample in the dataset, that includes data for a complete trip, contains 9 columns, that are as follows: Trip Id, Call Type, Origin Call,Origin Stand,Taxi Id, Timestamp, Daytype, Missing Data and Polyline. The Polyline column in each sample contains the list of GPS coordinates that a taxi traverses in a single trip. The

Figure 4.2: Taxi Dataset HeatMap

format of the Polyline data is [(latitude, longitude)], which in the dataset is represented as a string. Only Polyline data from the dataset is used in this study to maintain fair evaluations with models such as Cluster Based HMM that only model GPS trajectories [8].

4.2.2 Map Data

In this study, our service placement experiments are conducted in the city of Porto. This is because, anonymised taxi trajectory data for the city of Porto is freely accessible. To conduct the study, Simonstrator requires map data in the form of Osm.pbf file extensions, which are files that contain compressed map data. This enables Simonstrator to render the map on its in built GUI. The osm map data for the city of Porto is freely accessible on openstreetmap.org as well as on planet.osm. The bounding box coordinates for the map data on the city of Porto are East = -8.70, West = -8.50, North = 41.250 and South = 41.050.

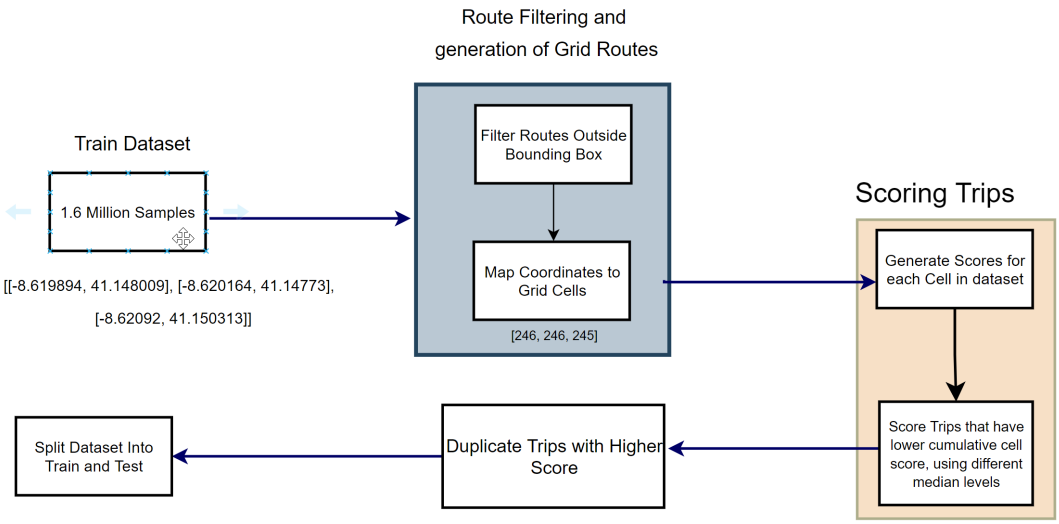

Figure 4.3: Dataset Pre-processing: High Level Overview

4.2.3 Pre-processing

The Taxi Trajectory dataset contains, more than 1.6 million records. However these records also contain trajectories that traverse out of Porto city. Figure 4.2 shows, the density of taxi trajectories in the city of Porto. It can be seen that there are large number of taxi trajectories that are spread out to areas beyond the limits of the city itself. Additionally, training the route predictions algorithms on the 1.6 million records is quite time consuming, hence the taxis are trained on a much smaller portion of the dataset, which is selected based on the density of the taxi trajectories in the area. Moreover, the dataset also contains routes that are more frequent in number, which leads to biased prediction performance. For instance, the bias in the dataset, will steer the route prediction algorithms to predict more frequent trajectories than routes that appear less frequently in the dataset. Figure 4.2 also shows the areas where the density of the taxi trajectories are low. Hence, to reduce the complexity and to remove any bias present in the dataset to a large degree, the dataset needs to undergo a few thoughtfully selected pre-processing steps.

To reduce the computational complexity of the route prediction algorithms trained on 1.6 million records, the map of Porto will be reduced to contain only the central region of the city. To extract the portion of the central region of Porto, QGIS desktop software was used. QGIS is an open source desktop software for manipulating Geographical Information data. Using QGIS, the map with the following bounding box, west =-8.6314, east = -8.5867, north = 41.16133, south = 41.1403, was extracted . After using QGIS for extracting the inner portions of the city, a reduced osm.pbf file is received, which is then configured into Simonstrator, to render the smaller map portion.

After extracting the required Map Data, the following steps are conducted:

- Route Filtering
  In this step, trajectories that have, GPS Coordinates that are outside the new bounding box of the internal city region are removed from the dataset. Fig 4.4 shows the heat map of the extracted taxi trajectories.

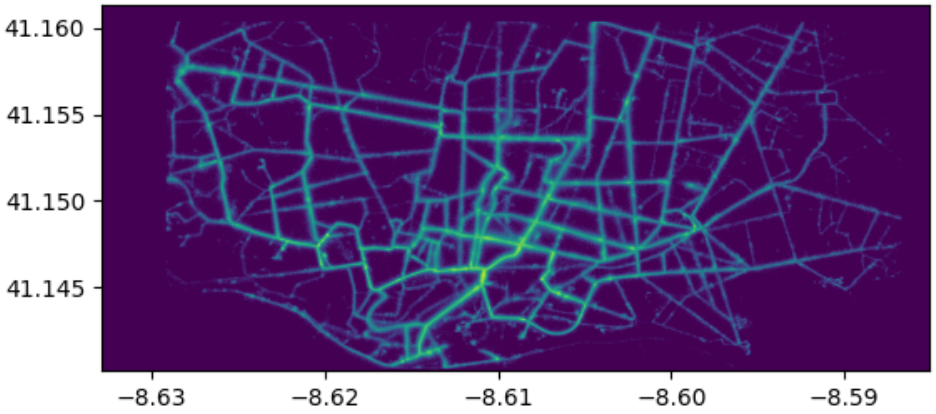

Figure 4.4: Extracted Taxi Heat Map

- Map Matching
  The GPS Coordinates in the trajectories are then mapped to Grid cells using a Grid Extractor class which will be discussed in later sections. Each grid cell has a unique identity, and hence the Taxi Trajectories are now composed of a list of Grid Cell identities. Furthermore, depending on the number of grid cells that the Map is divided into, the area that the GPS coordinate is mapped to can be quite big. This is because the taxi trajectories that are present in the dataset still collect, GPS coordinates, even if the Taxi remains stationary or is travelling slowly. Due to this the List will contain, multiple duplicate Grid Cell identities, which is not ideal for route prediction, as it can lead to a lot of bias as well as repetitive or uneven predictions. Hence, to improve the prediction performance of the route prediction algorithms, duplication of the grid cells ids is reduced to only 2 consecutive repetitions. For

instance, Grid Cells such as [45,45, 45, 46,46,46,46,47,47,47,48,48,49] are reduced to [45, 45, 46, 46, 47, 47, 48, 48, 48,49]. In this way the data can still capture slower movement between grid cells, while maintaining strong prediction accuracy.

- Scoring of Trips
  This step is conducted to remove any bias in the dataset, which maybe present due to the taxi movement in areas that are traversed more frequently. To achieve an unbiased dataset, in this dissertation a unique algorithm for bias removal in Trajectory data sets is implemented. First, each cell in the trajectory dataset is given a score based on the cumulative frequency of the cell calculated using all the trips present in the dataset. Next, the list of the grid cell scores are sorted and the 10^{th}, 20^{th}, 30^{th}, 40^{th}, 50^{th}, 60^{th}, 70^{th} and 80^{th} percentile of these scores are recorded. Then if 50% of a trip's grid cells scores, fall below any of the previously calculated percentiles, the trip is awarded a score based on the lowest percentile that trip falls into. These scores are awarded in descending order, so a trip that falls into the 10^{th} percentile will be awarded a score of 9 and if the trip falls into the 80^{th} percentile it will be awarded a score of 2. All the other trips, that do not fall into any of the percentiles mentioned, that is trips that fall into 80-100 percentile range, are awarded a score of 1.

- Trip Duplication
  From the previous step, we get a score for each trip. Based on that score the trip is duplicated in the database. For example, if a trip is given a score of 2, it will duplicated once, and for a score of 9 it will be duplicated 8 times. By following this approach, we are capable of removing most of the bias in the dataset, by adding more trips with cells that are less frequently visited. From figure 4.4, we can see varying levels of taxi trajectory density, in the new extracted map data. To account for this, Multiple percentiles were required to individually tackle different regions of the city that were traversed less. This is also evident when we see varying ranges of frequencies (scores) for different grid cells ranging from 360+ to grid cells with scores below 10. Hence by replicating trips based on the different percentiles, trips that had a very low score could be duplicated accordingly.

4.2.4 Map Matching

As discussed in previous sections, the Map Matching technique used in this study, is Grid Based Map Matching. Here, the Map is equally divided into N grid cells, where each region represents a spatial region in the map. In this dissertation, the python programming

language is used, along with the numpy library, to implement the Map matching technique, which is modelled as a Class with the name of GridExtractor.

The grid extractor class is initiated, with the following parameters. These parameters are the east, west, south and north bounding box GPS coordinates of the map region as well N, that represents the number of regions that the latitude and longitude range, will be divided into. There are 3 main methods that have been implemented in the Grid Extractor class.

The First method creates the Grid Map using the parameters provided while initiating the class.

```
def createGrid(self):
    N = self.N
    row_width = (self.north-self.south)/N
    column_width = abs((self.west-self.east)/N)
    count = 1

    # Upper Limit Calculation of the Row and Column Width
    for i in range(1, N+1):
        self.column_list.append(self.west + i*column_width)
        self.row_list.append(self.north - i*row_width)

    # Identity calculation for unique combination of row and column widths.
    for i in range(1, N+1):
        for j in range(1, N+1):
            self.grid_dict[(self.north - i*row_width,
            self.west + j*column_width )] = count
            self.id_dict[count] = (i, j)
            count = count+1
```

The grid map method divides the latitude and the longitude range in to equal widths. Each width is identified by the upper limit of that width's GPS Coordinate. Using this a column list and a row list are created that uses the GPS identities of the width for the latitude and the longitude respectively. Finally, every combination of the row and the column width are given a unique identity and are stored in the python dictionary.

Next, the getGridBlock method, is used to compute the unique identify of the grid block given the GPS Coordinate.

```
    def getGridBlock(self, lat, lon):
        out_of_bounds = self.checkLatLngBounds(lat, lon)
        if out_of_bounds == False:
            return -1
```

```
        if lat <=self.row_list[0] and lat>=self.north:
            lat = self.row_list[0]
        if lon >= self.column_list[0] and lon <= self.west:
            lon = self.column_list[0]

        for i in range(0, self.N -1):
            if lat <self.row_list[i] and lat>=self.row_list[i+1]:
                lat = self.row_list[i+1]
            if lon > self.column_list[i] and lon <= self.column_list[i+1]:
                lon = self.column_list[i+1]
        return self.grid_dict[(lat,lon)]
```

To map a GPS coordinate to a Grid Cell, the given latitude and the longitude are compared against the respective width ids stored in the row and column list. If a coordinate lies between index i and i+1 of the column and row lists, then i+1 width id is returned. This is then used to access the Grid Cell id from the Grid_Dict dictionary.

Finally, to get the Coordinates of the Grid Id, that a GPS Coordinate was previously mapped to, the getGridCenter is method is used.

```
    def getGridCenter(self, id):
        rowid,columnid= self.id_dict[id]
        row_width = (self.north-self.south)/self.N
        column_width = abs((self.west-self.east)/self.N)

        lat = ((self.north - (rowid-1)*row_width) + (self.north - rowid*
    row_width))/2
        lon = ((self.west + (columnid-1)*column_width) + (self.west +
    columnid*column_width))/2

        return lat, lon
```

In this method the centers of the Grid cells are returned, by computing the midpoint of the upper and lower limit of the GPS coordinates for each of the respective row and column widths.

Using these three methods, Grid Based Map Matching is performed in this study.

4.3 Route Prediction

In this section, we discuss some of the important implementation techniques of the route prediction algorithms discussed in previous sections. We will cover the implementation details for the Sequence to Sequence models and the Hidden Markov Model. Since, the Cluster based Hidden Markov Model was not implemented in this study, the details about the implementation of that algorithm will not be discussed here.

4.3.1 Sequence to Sequence Models

The RNN Seq2Seq Models have been implemented using the Python programming language. The model itself has been built using Pytorch, to automatically generate the model's trainable network. Other popular libraries like Tensorflow, were also experimented with, however, compared to the these libraries, pytorch supports easily understandable syntax and has superior support available online. Furthermore, due to these advantages, pytorch is more popular for academic work.

The next sections will discuss some the methods used for implementing the Sequence to Sequence model and the Attention based Sequence 2 Sequence model.

Standard Sequence to Sequence Model

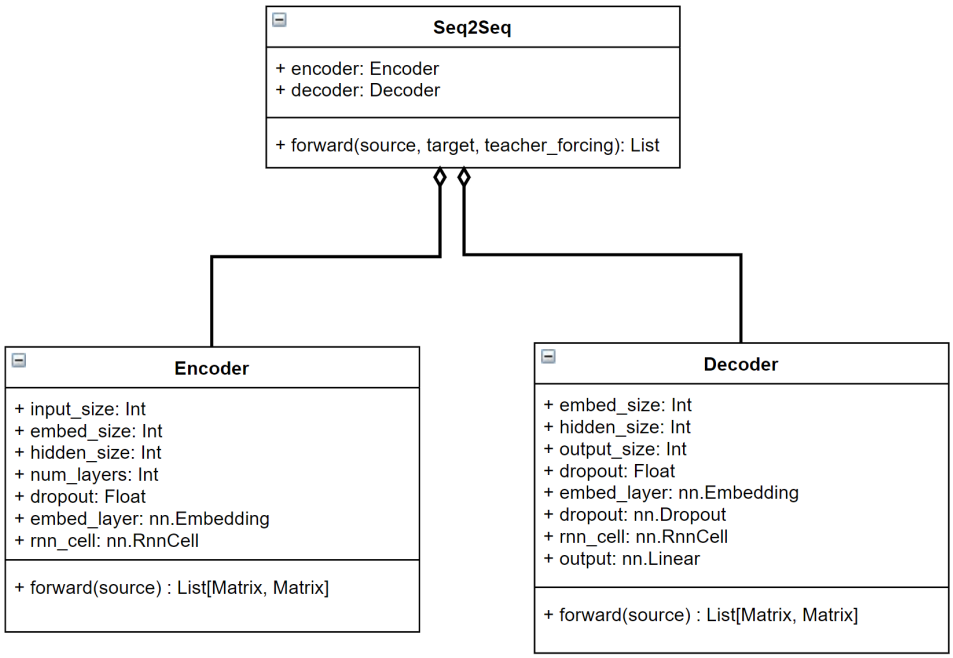

Figure 4.5: UML Diagram: Sequence to Sequence Model

The standard sequence 2 sequence model, is implemented using three separate classes as shown in Figure 4.5. These classes are, the Encoder, Decoder and the Seq2Seq class. Each of these classes extend the nn.module class from pytorch, that internally builds the trainable Network structure for the model.

Encoder Class

The Encoder Class is responsible for converting the input sequence into a fixed size vector. The inputs to the encoder class are the size of the input sequence, the embedding size, the size of the hidden state, the number of layers in the encoder network and dropout. The encoder class contains the dropout layer, Rnn Cell, Output Layer and the Embedding Layer. Table 4.2 describes the inputs to the encoder class and table 4.3 describes the layers present in the encoder class.

Encoder Inputs	
Input Name	**Description**
Embedding Size	The embedding size is used by the embedding layer in the encoder to convert the input sequence into vector representations. The decoder and encoder embedding size is the same, since the inputs belong to the same vocabulary of categorical values.
Hidden Size	The hidden size is used to configure the size of of the internal states of the RNN cell. These internal states can either be the Hidden State if a GRU cell is used or both the hidden and the cell state, if a LSTM Cell is used.
Input Size	The input size represents the the number of categories present for each grid cell prediction. This is equal to the Number of Grid Cells, that the map is divided into.
Dropout	Represents the percentage of the weights in the encoder network that are forced to zero to reduce over fitting.

Table 4.2: Description of the inputs to the Encoder Class

Encoder Layers	
Layer Name	**Description**
Embedding Layer	The embedding layer is used to convert the input sequence in the encoder network to embedded vectors.
Dropout	The dropout layers is applied to the RNN Cell in the encoder network to reduce overfitting.
RNN Cell	This configures what kind of RNN cell is used in the encoder Network. In this dissertation, we implement both a GRU and a LSTM based Encoder Network.

Table 4.3: Description of the Layers in the Encoder Class

Decoder Class

The decoder class is responsible for decoding the fixed size vector, along with inputs to the decoder, to the target sequence. The inputs to the decoder class are the embedding size, the hidden size, target size and dropout. The decoder contains an embedding layer, a dropout layer, the Rnn Cells at each layer and a output feed forward neural network. Table 4.4, describes the inputs to the decoder network and table 4.5 describes the layers in a decoder network.

Decoder Inputs	
Input Name	**Description**
Embedding Size	The decoder embedding, is used by the embedding layer in the decoder to convert the target sequence inputs to vector representations. The decoder and encoder embedding size is the same, since the inputs belong to the same categorical vocabulary.
Hidden Size	Same as the encoder network, the hidden size is used to configure the size of of the internal states of the RNN cell.
target Size	The target size represents the the number of categories of the Grid Cell in the decoded Sequence.
Dropout	Represents the percentage of the weights in the decoder network that will be removed to reduce over fitting.

Table 4.4: Description of the inputs to the Decoder Class

Decoder Layers	
Input Name	**Description**
Embedding Layer	The embedding layer is used to convert the target inputs to the decoder into embedded vectors.
Dropout	The dropout layers is applied to the RNN Cell to reduce over-fitting.
RNN Cell	This is used to configure which type of RNN Cell is used to train the decoder network.
Output Layer	The Output Layer is a feed forward neural network that converts the output of the hidden state from the decoder RNN cell to a vector of size equal to the number of unique grid cells.

Table 4.5: Description of the Layers in the Decoder Class

Seq2Seq Class

The seq2seq class, integrates the encoder and the decoder classes together. The forward method in the seq2seq class conducts the follows steps:

- The forward method is called with the input sequence and a target sequence. During training the real target sequence will be used.
- The input sequence is then fed into the encoder and a fixed size context vector is returned.
- Next the decoder class's forward method is called for each value in the target sequence and the context vector. To signal the start of decoding at timestep 1, the first data input to the decoder is a 1.
- The output of the decoder is the predicted value of the target sequence at time t. Also, the hidden state and the cell state of the decoder at timestep t is returned.
- The decoded vector, is then passed through an argmax function, which finds the index of the vector with the highest probability. This index corresponds to the identity of the grid cell.

Using these steps the model is capable of decoding the required target sentence. While training the Seq2Seq model a method called teacher forcing is used. The technique feeds the decoder at timestep t+1, with either the correct next target value or the value decoded by the decoder network during the previous timestep. The decision of feeding the correct input of the predicted input, is based on a random function. Using this technique the model is forced to predict correct values, which speeds up the training process.

Attention Based Sequence to Sequence Model

The Attention Sequence 2 Sequence model, was implemented using four separate classes. Similar to the Standard Seq2Seq model, the four classes extend the nn.module class from pytorch. Figure 4.6 shows the four different classes used in the Attention Sequence 2 Sequence model. As compared to the Standard Seq2Seq model, this model has another class called, attention, that is responsible for the calculation of the alignment scores, used in the context vector.

The attention class, calculates the alignment scores using the hidden state of the previous decoder time step and the encoder hidden states, which is passed through a feed forward neural network and a softmax function. These alignment scores are then returned to the decoder class, where the new context vector is generated using the current input

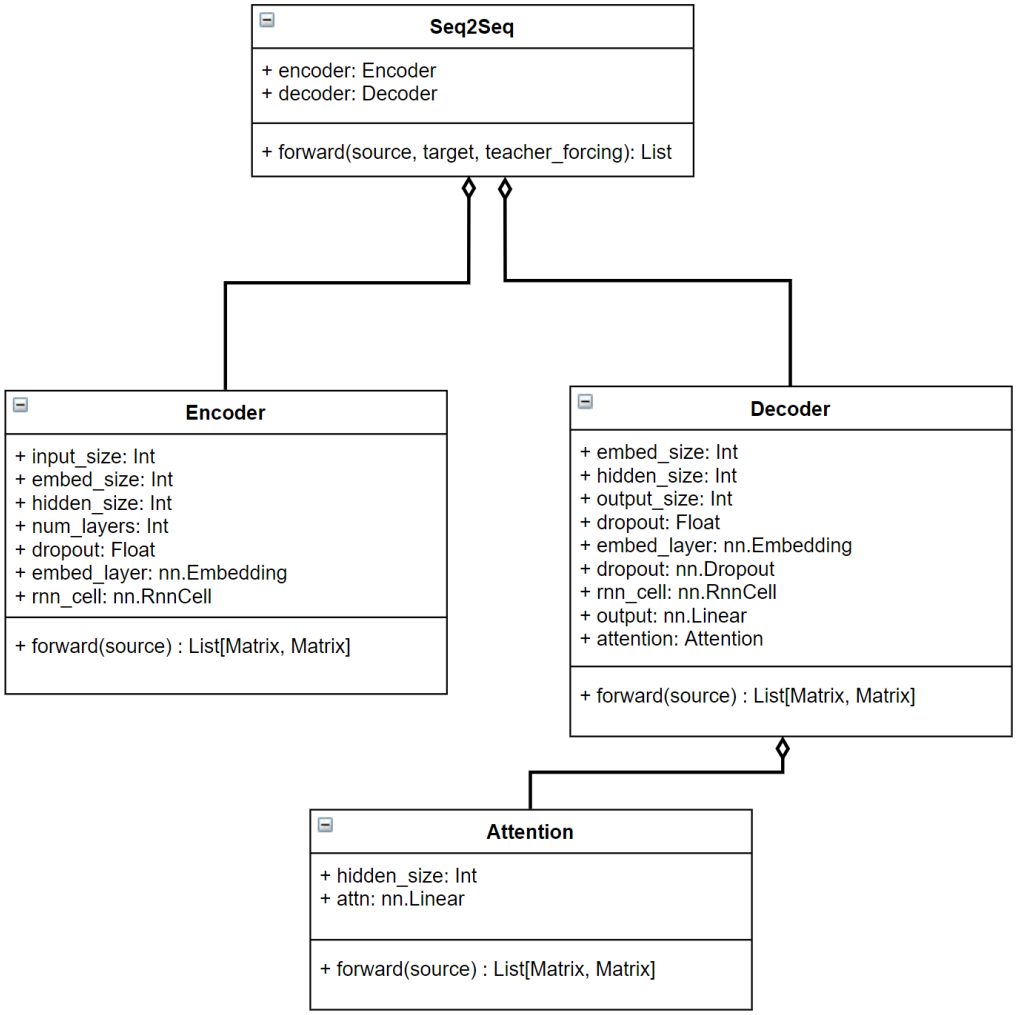

Figure 4.6: UML Diagram: Bahdanau Attention based Sequence to Sequence Model

to the decoder. This entire process is missing in the standard Seq2Seq model, which differentiates it from this implementation of the attention model.

The rest of the steps outlined in the Standard Seq2Seq model are identical to the attention model.

Training Dataset Preparation

Before training a sequence 2 sequence models, data required for training the model needs to be prepared to match the requirements of the input sequence length Il and the target sequence lengths Ol. The input and target sequence need to be extracted from the dataset, according to their respective sizes. While the input and the output sequences are completely independent from each other, in this dissertation, to reduce the complexity of the system, we use the same sequence lengths for both. The sequences are extracted by

looping over all the routes in the dataset, and using a sliding widow, of length (Il + Ol), over an entire route . In this way the actual trainable dataset can be extracted.

Furthermore, the values inside the input and the target sequence can be variable. For instance, if the max length of a input sequence is 8, then the number of real data points inside this sequence can range from 1 to 6. The remaining values are represented by a start and stop symbol. We use 1 as our start symbol and 0 as our stop symbol. Using these symbols a sequence of size 6 and actual data entries of size 2 can be represented as [1, 34, 35, 0, 0, 0]. While creating the dataset, we also make sure to vary the lengths of the data points in both the input and the target sequence, to handle prediction requests with variable sizes.

4.3.2 Hidden Markov Model

In this section some of the important implementation details of the Hidden Markov Model will be discussed. Since, the implementation of this HMM relies on custom statistical modelling properties, it is hard to rely on any external frameworks for model building, like in the case of Sequence 2 sequence networks. Hence, this HMM model has been implemented using just Python and the Numpy library.

The HMM model has two states, the Observed States O_i and the Hidden State H_i. Unique fixed partial size routes represent the hidden states in this model. From preliminary testing, partial routes of size 2, seem to achieve the best performance, and any size greater than 2 degrades the model capabilities further. Hence, size 2 partial routes are used as the hidden states in this Markov Model for further testing and comparative analysis.

An object oriented approach is used to build the HMM model with the help of a python class. This class has the following methods:

def get_transition_prob()

This method is responsible for building the transition probability Matrix A of size (N x N), where N is the number of unique hidden states in the model. Since, the hidden states are partial route sequences, a sliding window of size 2 is used first on all the routes in the datasets, to retrieve all the unique partial routes. Next, a Transition Frequency dictionary is created to store the frequency of transitioning from one partial route i to (i+1). Sequence i and i+1 can be easily acquired while traversing the same loop used to retrieve the unique partial routes. The transition frequency dictionary is then used to calculate the transition probability matrix, by dividing the frequency of transitioning from Hidden State i to (i+1) by the total frequency of transitioning from hidden state i to all the hidden states.

def get_emission_prob()

The get_emission_prob method is used to calculate the emission probability matrix B, of size N x M, where M is the number of Observed states. This method captures the probability of observing state O_i in Hidden State H_i. We have discussed that each hidden state is a unique partial trajectory from the taxi trajectory dataset. To calculate the emission probability, the number of frequency of cells in each Hidden State is divided by the number of cells in that Hidden State.

$$H_i = [0_1, 0_2] \qquad \text{Partial route of size 2}$$

$$B[i, 1] = \frac{Number\ Of\ O_1\ in\ H_i}{Length\ of\ H_i}$$

def get_initial_prob()

To calculate the Initial Probability Matrix π, of Size Nx1, the get_emission_prob method is used. We calculate the initial probability by dividing the frequency of Hidden State H_i in the dataset, by the total frequency of all the hidden states.

def viterbi()

The viterbi method is used to calculate the future path of the user, with the help of the Viterbi algorithm. As discussed in the previous chapter, the Viterbi algorithm decodes the best hidden state sequence that describes the sequence of observed states provided. However, the standard viterbi algorithm can only describe the sequence of hidden states up til time t. Hence, we modify the viterbi algorithm to predict the hidden state for time t+1 as well. The algorithm also depends on the transition probability matrix (A), the Emission Probability Matrix (B), and the Initial Probability matrix (π). Algorithm 3 provides the pseudo code for the Viterbi algorithm implemented in this dissertation.

Algorithm 3: Viterbi Algorithm

Data: Observation Sequence O

Result: Matrix of Size (Length(O)+1) Predicted Sequence

L = Length(O);

Initialize N x (L+1) Viterbi Matrix V;

Initialize N x (L) BackTracking Matrix Bt;

Initialize (L+1) x 1 Predicted Sequence list St;

begin

  $\forall_{i=1}^{N}$V(i,0) $\leftarrow \pi \circ B(i, O^1)$;

  for *n from 1 to L+1* **do**

    for *i from 0 to N* **do**

      if *i* $\leftarrow$ *(L+1)* **then**

        V(i,n) $\leftarrow max_{i'=1}^{n}$(V(i', n-1) $\circ$ A(i', i)) $\circ$ B(i, O^n)

      end

      else

        V(i,L+1) $\leftarrow max_{i'=1}^{n}$(V(i', n-1) $\circ$ A(i', i))

      end

      Bt(i, n-1) $\leftarrow$ argmax$_{1 \leq i' \leq N}$(V(i', n-1) $\circ$ A(i', i))

    end

  end

  St(L+1) $\leftarrow$ argmax$_{1 \leq i \leq N}$(V(i,-1));

  for *j from L to 0* **do**

    St(j) $\leftarrow$ Bt(St(j + 1), j)

  end

  Return St

end

This Viterbi algorithm differs from other standard viterbi algorithms by predicting the Hidden Sequence of States up till time t+1. This is achieved by adding an extra step during the induction step to calculate the viterbi value at time t+1. However, since, this step is greater than the length of observation sequence, the final Viterbi value is calculated without the Emission Probability. Equation 4.1 shows the step in the viterbi algorithm, that calculates the Viterbi value for Hidden State H_i for Observation O_j. Equation 4.2 shows the added step to the viterbi algorithm

$$V[i, n] \leftarrow max(A[:, i] \circ V[:, n-1]) * B[i, O(n)] \tag{4.1}$$

$$V[i, L+1] \leftarrow max(A[:, i] \circ V[:, n-1]) \tag{4.2}$$

By adding this extra step to the Viterbi algorithm, we can calculate the next Hidden State. We can also predict more than one hidden states by increasing the number of prediction timsteps. However, preliminary testing has shown poor performance when the timesteps are increased. Hence to predict future grid cells of length greater that 1, the viterbi algorithm is re-un for every new additional grid cell prediction. This method of re-running the viterbi algorithm for predicting future hidden states performed better and hence was added to our algorithm for the final evaluation.

4.4 Summary

In summary, this chapter focused on the implementation process of the algorithms relevant to our study. The chapter introduced the environments in which the different algorithms were implemented. Then a detailed description of the pre-processing steps conducted on the Taxi Trajectory Dataset, to prepare the data for route prediction and service placement simulation, was given. Finally, the the implementation details for each of the route prediction algorithm was discussed. In the next section, these route prediction algorithms will be evaluated individually and in the context of the service placement algorithm.

Chapter 5

Evaluation

This chapter describes the evaluation process and details the results of the route prediction algorithms used in this dissertation. Similarly, performance of the route prediction algorithms in the context of the service placement algorithm, will be discussed here.

5.1 Objectives

The primary objective of this chapter, is to understand, whether route prediction algorithms can help improve the performance of a service placement algorithm. To achieve that objective, first the individual performance of the route prediction algorithms, have been acquired and are compared to form a preliminary understanding of what algorithms can help the service placement algorithm better. Using this preliminary knowledge, specific versions of the algorithms are selected for further analysis in the context of service placement. Finally, these route prediction algorithms are integrated with an Ant Colony Optimization based Service Placement algorithm and any sign of performance improvements are noted.

5.2 Route Prediction Experiments

The evaluation process and the testing of each of the route prediction algorithms individually, will be discussed in this section. Two main evaluation metrics are used to compare the different route prediction algorithms. These are the Haversine Distance and the Accuracy of the model. Additionally, while training the Sequence to Sequence models, Categorical Cross-Entropy loss is used for gradient descent. This loss function will also be utilized to evaluate between the different sequence 2 sequence models.

Haversine Distance

Haversine Distance is the measure of the great circle distance between two coordinates on a sphere. This formula has been popularly used to calculate distances between 2 points on the earth, by specifically using the radius of the earth as a parameter in the formula [7]. The haversine distance can be described using the formula given below [7]:

$$d = 2rsin^{-1}(\sqrt{sin^2(\frac{\phi_2 - \phi_1)}{2} + cos(\phi_1)cos(\phi_2)sin^2(\frac{\lambda 2 - \lambda 1}{2})}) \tag{5.1}$$

where, ϕ_1 and ϕ_2 are the latitudes and λ_1 and λ_2 are the longitudes between two positions in radians and r is the radius of the earth in Kms.

The haversine formula is used in this dissertation to calculate the distance between the predicted route and the actual route. The order of the route is not prioritized in this calculation as we are majorly concerned with the estimating the distance of the predicted route from the edge servers closest to the future path of the user. If the haversine distance is large between the predicted route and the actual route, then the model will be considered to perform relatively poorly.

Accuracy

Accuracy can be defined as the ratio of the total number of correct prediction, by the total predictions.

$$Accuracy = \frac{Total\ number\ of\ Correct\ Predictions}{Total\ number\ of\ Predictions}$$

In this calculation the order of the predicted route matters, as the cell values of the predicted and actual route are compared based on their indexes. The higher the accuracy the better is performance of the model.

Categorical Cross-Entropy Loss

Categorical Cross-Entropy loss is used to perform gradient descent on predictions with multiple categories. In our study, each cell prediction, can take the value between 1 to N number of Grid Cells. This loss function was chosen, as it converts the output of the Seq2Seq network to probabilities before calculating the loss. It does this by using a softwax function over the values produced by the Seq2Seq network. It can be described using the formula given in Equation 5.2 [35]:

$$loss(y, class) = -log(\frac{\exp(x[class])}{\sum_j \exp(x[j])}) \tag{5.2}$$

=While comparing the results of the different models, the model with a lower loss would be considered to perform better.

Other evaluation techniques for the route prediction models that have been used are:

- Varying Grid Maps
  Each model is trained on taxi trajectory data mapped to Grid Maps with varying sizes. For this study we compare the performance of the model when trained on Grid Maps with 100, 225 and 400 Grid Cells.

- Varying Prefix Cells
  We also evaluate the performance of all the models on partial route trajectories of varying size. That is, these models are evaluated on partial routes that contain t = 1 to 6 grid cells, after map matching. The aim is to observe whether an increase in performance arises due to the use of a bigger partial route sequence.

- Varying Cell Prediction
  Finally, only the Hidden Markov Model, will be evaluated using route predictions of varying sizes. That is, the performance of model will be judged by the number of cells the model predicts. This evaluation technique is only used in the Hidden Markov Model due to the flexibility with which the Viterbi Algorithm is used to predicted future cells. The future cells predictions cannot be controlled effectively for the sequence to sequence and the Cluster based HMM.

To reduce bias during comparisons made in this dissertation, the models are trained and evaluated on the same data. Since the execution time of each algorithm varied drastically, and due to the time constraint of this dissertation, our study uses only the three metrics given above to evaluate the models, and these models are only trained on 10000 taxi trajectory routes.

The next sections, will describe the results acquired by the various route prediction models studied.

5.2.1 Sequence to Sequence Algorithms

In this section, we compare the performance of the different sequence to sequence models implemented in this dissertation. Each model is run for 10 epochs and is then compared based on the different evaluation metrics stated in the previous section.

Table 5.1 details the configuration parameters used to train the sequence to sequence models.

Configuration	Value
Batch Size	128
Number of Epochs	10
Learning Rate	0.0001
Input Size	8
Output Size	8
Hidden Size	512
Number of Layers Encoder	2
Number of Layers Decoder	1

Table 5.1: Configurable Variables for the Sequence to Sequence Models

5.2.1.1 Sequence to Sequence Training Results

This section will discuss the results of the Sequence to Sequence models evaluated individually. The initial results will contain the average haversine and validation Categorical Cross Entropy loss for each epoch of training. Furthermore, the final average haversine and validation loss values will be plotted to compare the performance of the different models after training. Finally, these models have been evaluated using three different grid map variations and the results for each variation have been displayed below.

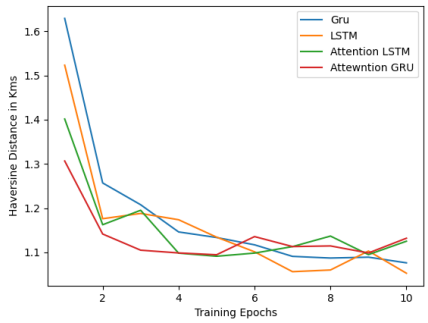

Figure 5.1: Haversine Distances for 100 Grid Cells

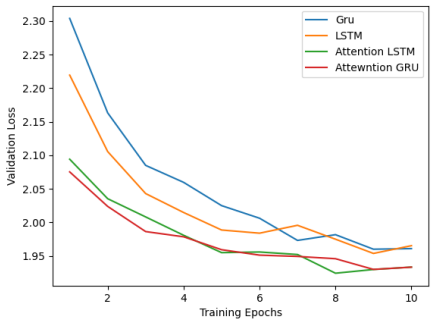

Figure 5.2: Validation Loss for 100 grid cells

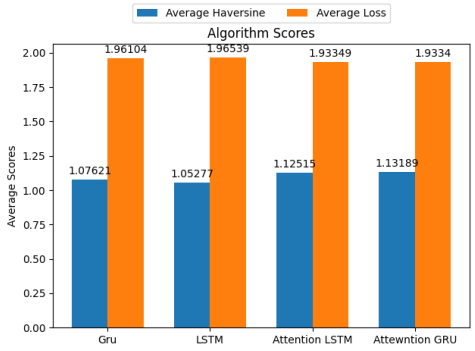

Figure 5.3: Average Validation and Haversine Distance 100 Grid Cells

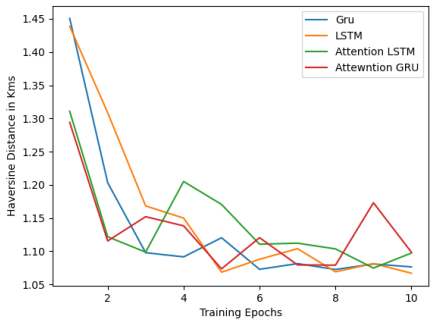

Figure 5.4: Haversine Distances for 225 Grid Cells

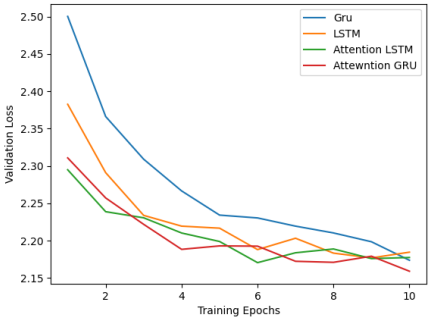

Figure 5.5: Validation Loss for 225 grid cells

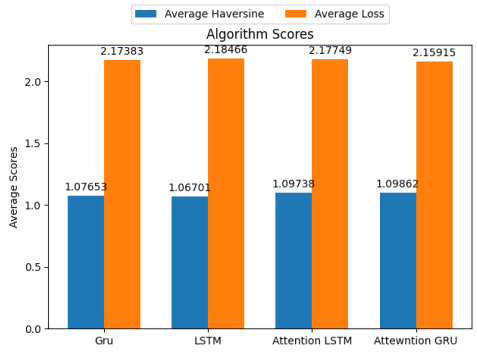

Figure 5.6: Average Validation and Haversine Distance 225 Grid Cells

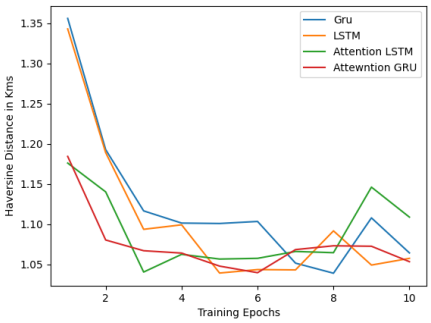

Figure 5.7: Haversine Distances for 400 Grid Cells

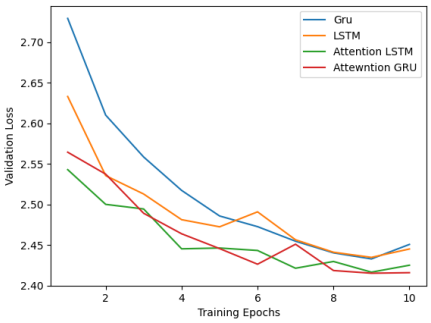

Figure 5.8: Validation Loss for 400 grid cells

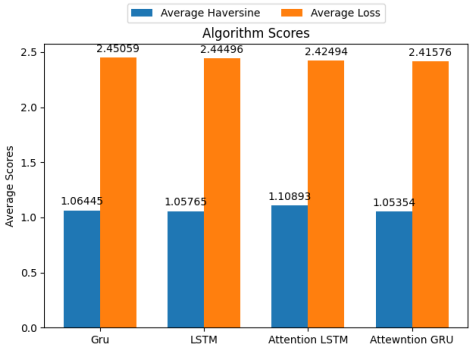

Figure 5.9: Average Validation and Haversine Distance 400 Grid Cells

Figures 5.1, 5.4, 5.7 display the average Haversine distance across all the seq2seq models for each training epoch. Similarly, figures 5.2, 5.5 and 5.8 display the average validations loss across all the models for each training epoch. Finally, the bar charts in 5.3, 5.6 and 5.9 display the average validation loss and Haversine distance of the seq2seq models after training.

We see that all the sequence 2 sequence models perform similarly, with variations seen in the different metrics used. To compare the performance of the sequence 2 sequence models with the other route prediction models, the Attention based Sequence to Sequence network with GRU Rnn cells is used, since it shows good consistent performance among the three different grid cells configurations.

5.2.1.2 GRU Attention Results

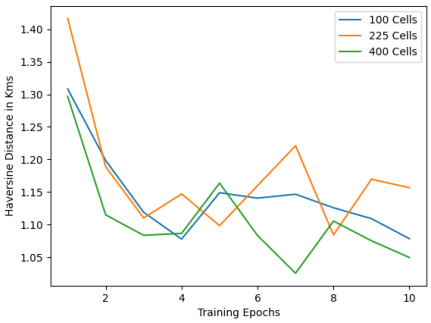

Figure 5.10: Haversine Distances (Different Grid Cells)

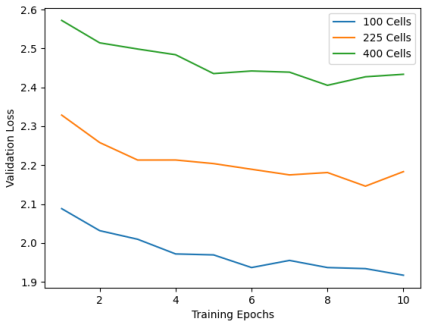

Figure 5.11: Validation Loss for Different Grid Cells

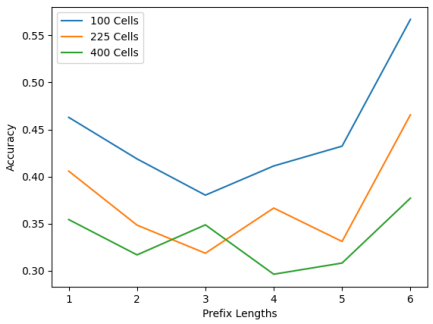

Figure 5.12: Accuracy for different Prefix Cells

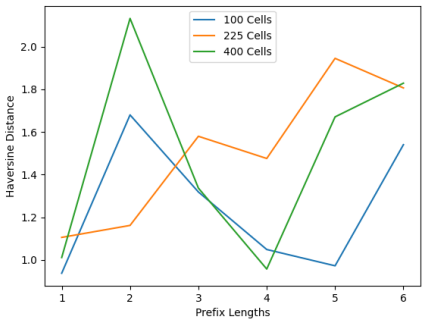

Figure 5.13: Haversine Distance for different Prefix Cells

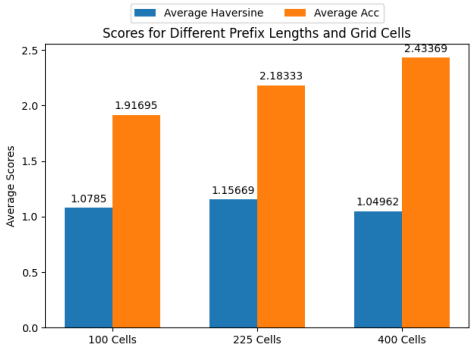

Figure 5.14: Average Scores for Different Grid Cells

The results in this section, capture the performance of the Gru Based Attention Model. In figures 5.10, 5.11 capture the performance of the model using different grid cells during training. We can see that the 100 grid cell model shows better performance in both

metrics. Furthermore, the barchart given in 5.12 captures the average Haversine and accuracy results of the model for all the different grid maps. These results clearly shows that the model trained on a 100 Cell Grid Map, performs better that the other variants.

Furthermore, this model is also evaluated on the performance of the model on prefix cells of varying sizes. This evaluation strategy is used, to compare the GRU Attention Model against, the other route predictions algorithms studied. Here the metrics used are the Average Accuracy and the Haversine Distance. Accuracy is used so that the GRU model can be directly compared with the other route prediction models in the dissertation, as Cross Entropy Loss is only used by the seq2seq models during training. Figures 5.13 and 5.14 displays the results of the models evaluated using the average haversine and accuracy metrics for different prefix lengths. From these results, we can observe that the 100 grid cell Gru Attention model still performs better than the other model variants.

However, what we don't witness, that we see in the other models in this study, is a discernible pattern in the performance of the model for different prefix routes. It is usually the case that a higher number of prefix cells (or a larger partial route sequence), should demand better performance from the models, as the model is provided with a greater context to make better predictions. A reason for this could be that neither attention based seq2seq models or the standard seq2seq model, show major differences in their performance. Hence, we can conclude that neither models, pay attention to the full input sequence and only learn mappings between the input and the output sequence. This could explain the lack of any improvement in the models, when a bigger partial route sequence is provided.

As we have seen in this section, the GRU Attention Based Seq2Seq model performs the best when trained on the Taxi Trajectories that have been mapped to a Grid Map containing 100 grid cells. This particular model will be used when we finally compare all the route prediction models studied.

5.2.2 Hidden Markov Model

The Hidden Markov model, will be evaluated using the average Haversine distance and accuracy of the model. Due to the flexibility of the viterbi algorithm discussed in the previous chapter, the performance of this model will be measured on varying number of prefix cells as well as the number of future cells predicted. Since the sequence to sequence model, uses a maximum of only 6 grid cells for prediction, the HMM model will also be evaluated on prefix cells ranging from 1 to 6 cells. Similarly as the Seq2Seq model also predicts a maximum of 6 future cells, the model will be compared based on t number of future cell predictions ranging from 1 to 6.

5.2.2.1 HMM Results

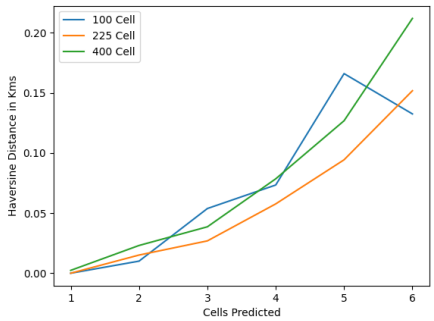

Figure 5.15: Haversine Distances for N number of Cells Predicted

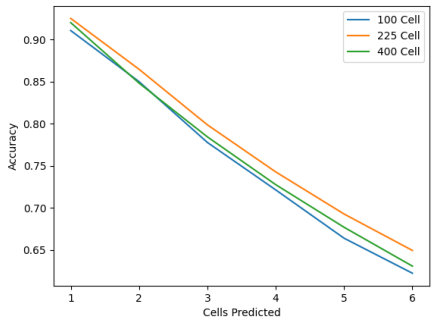

Figure 5.16: Accuracy for N number of Cells Predicted

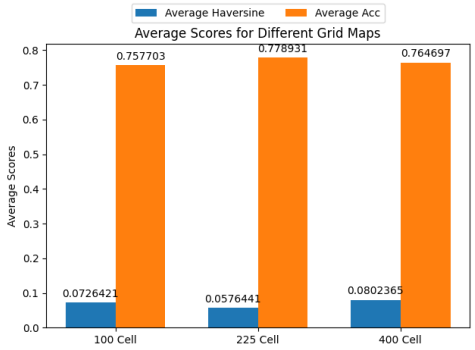

Figure 5.17: Average Scores for N number of Cells Predicted

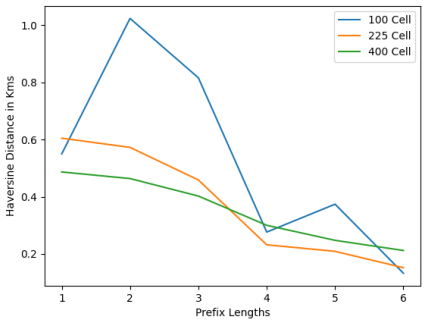

Figure 5.18: Haversine Distances for N number of Prefix Cells

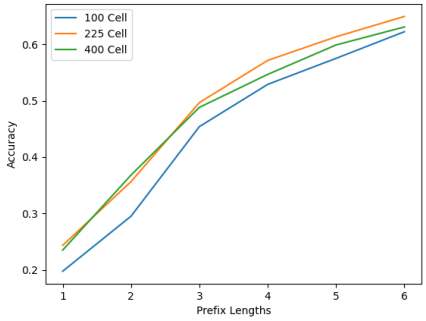

Figure 5.19: Accuracy for N number of Prefix Cells

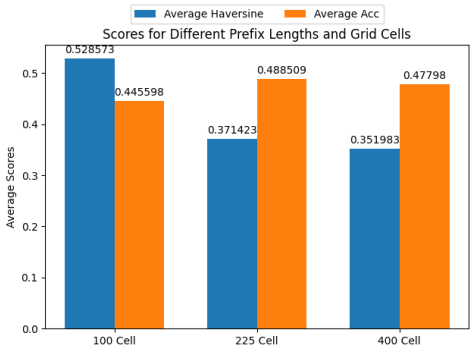

Figure 5.20: Average Scores for N number of Prefix Cells

Figures 5.15, 5.16 and 5.17, plot the performance of the HMM model for varying number of cells predicted. For these results the models were evaluated using a constant number of prefix cells, 6 in this case. The models show almost zero haversine distance and accuracy of above 90% for predicting only one grid cell. When the we increase the number of grid cells predicted, the model naturally shows a decrease in performance, when compared to

the seq2seq models. When six grid cells are predicted, the accuracy falls down to around 65% and the haversine distance increases to about 200 meters. This discernible pattern that we notice in the Hidden Markov Model is due to the decrease in performance when the viterbi algorithm is re-used for every new prediction.

Next, figures 5.18, 5.19 and 5.20 describes the performance of the model for varying number of Prefix Cells. For this evaluation, we fix the number of cells predicted to 6. As the HMM model predicts 6 future Grid Cells, we see the performance dips quite a bit, as the prefix lengths start with only 1 and then rise to 6 cells. This shows that the model requires a higher number of prefix cells to show better performance. Again, the number of cell predictions and prefix sizes were taken to compare the HMM model against the seq2seq model, but has the flexibility to predict or use a higher number of cells if required.

From the various results that we have discussed in this section, we notice, that the HMM model built using a Grid Map with 225 Grid Cells performs the best. This model will be used later for future comparison and performance evaluations. We also notice that the HMM model, performs much better that the Sequence 2 Sequence model. This observation will be discussed in the next few sections.

5.2.3 Cluster Based Hidden Markov Model

The Cluster based Hidden Markov model, will be tested on varying prefix cells. Similar to the HMM model, prefix cells ranging from 1 to 6 will be used to evaluate the performance of this model. However, this model is not capable of flexible future route predictions, as the model predicts the cluster that the partial route belong to and then extracts the full route of the Taxi, from that cluster. Hence, it is not evaluated on varying grid cell predictions.

5.2.3.1 Cluster Based HMM Results

In this section, the results of the Cluster Based Hidden Markov model will be discussed.

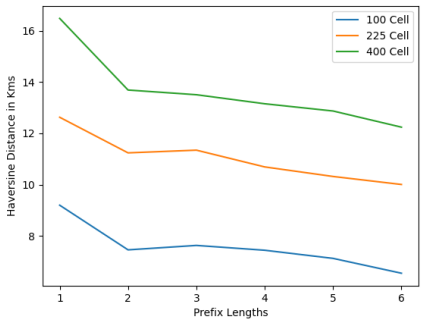

Figure 5.21: Haversine Distances for N number of Prefix Cells

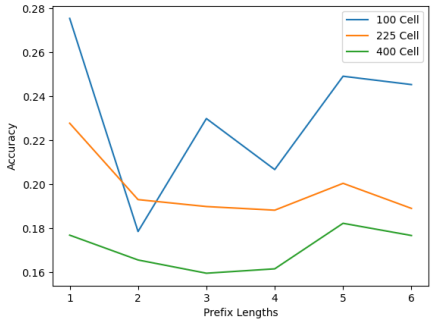

Figure 5.22: Accuracy for N number of Prefix Cells

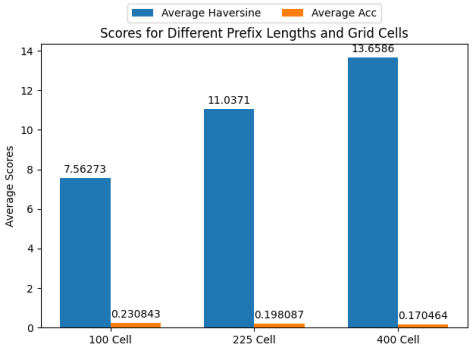

Figure 5.23: Average Scores for N number of Prefix Cells

Figure 5.21 shows the performance of the model, for varying number of prefix cells, using the haversine distance metric. We witness superior performance by the model trained on 100 grid cells for this metric. Similarly, in figure 5.22 that plots the accuracy of the model, we again observe that the 100 grid cell model performs better than the rest. The average scores for each of the models shown in the figure 5.23 makes this performance distinctions even clearer.

Hence, for future comparisons, the Cluster Based Hidden Markov Model trained on a Grid Map containing 100 Grid Cells will be used.

5.2.4 Route Prediction Discussion

Here we will compare the best models from the different route predictions algorithms discussed in the previous sections. These models that will be compared in this section are the GRU Based Attention Model trained on a 100 Cell Grid Map, a HMM trained on a 225 Cell Grid Map and a Cluster Based HMM trained on a 100 Cell Grid Map. Since we have used prefix cells as an evaluation strategy in all the algorithms, the same graphs for the best models will be plotted together and compared. Also, to clarify the

performance of the models when compared with each other a bar graph will used that shows the average scores for the different models together.

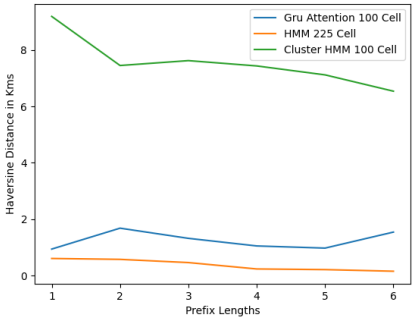

Figure 5.24: Haversine Distances for N number of Prefix Cells

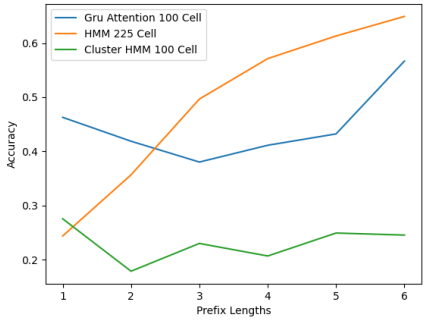

Figure 5.25: Accuracy for N number of Prefix Cells

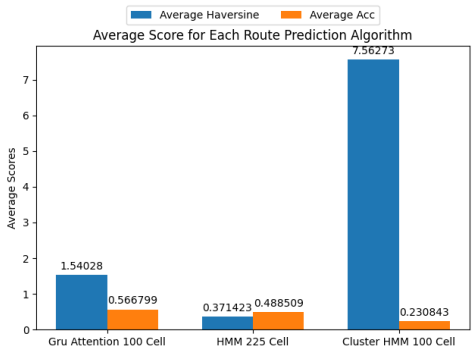

Figure 5.26: Average Scores for the Route Prediction Algorithms

The plots for the three models is given above. Figure 5.24, compares the average haversine distance, for varying prefix cells for the selected different models. From this figure, we see that the HMM and Gru Attention based Network perform better than the Cluster Based HMM implemented in [8] and [19]. Similarly, we see the same performance gap when compared and evaluated using the accuracy metric, seen in figure 5.25. Furthermore, we notice the distinction in performance more clearly in the final figure 5.26.

So we can confirm that in this study, two different prediction algorithms were implemented that successfully outperformed the Cluster Based HMM implemented in previous work. This is a step forward and better results from the Service Placement Algorithm are expected from the two new models, which will be discussed in the next section. Also, these results also confirm that the Hidden Markov Model using the Viterbi algorithm for prediction, performs slightly better than the GRU based attention model. Using this observation, it can be noted that a similar improvement in performance between the Gru Attention Model and HMM is expected in the Service Placement Results. In the next

sections, we will discuss some of the experiments conducted on the Service Placement algorithm and will review the results acquired.

5.3 Service Placement Experiments

This section, will dive into the experiments conducted for measuring the performance of the ACO based Service Placement Algorithm integrated with the route prediction algorithms selected in the previous sections. The configurations used to run the simulation experiments and the metrics used to evaluate the performance of the service placement algorithm will be described int the next few sections. Finally, based on this performance, the route prediction models will be compared.

5.3.1 Experiment Setup

In this section, the configuration parameters used to run the experiments on the simulator are described. Table 5.2 captures the configurations parameters that the Simonstrator simulation framework experiments are conducted on. Table 5.3 describes the configurations of the ACO based Service Placement Algorithm.

Configuration	**Value**
Number of Cloud Servers	1
Number of Edge Servers	100
Placement of Edge Servers on the map	Grid Based
Number of Taxi Users	1
Taxi Service Request Arrival Time	Ever 2 Minutes

Table 5.2: MEC System Configuration

The MEC system is configured to have 1 Cloud server and 100 Edge Servers. These Edge servers are placed in different locations following on a grid layout. For instance, at the center of every grid in a Grid Map a Edge Server will be placed. Next, the experiments are run on only one taxi user at a time. Previous work used simulated user movement to move users between randomly generated locations. However, since this dissertation is tested on Real Taxi Trajectory data, the csv data for each taxi had to be loaded in. Currently, the extra configurations, that would be required to run multiple taxis in the simulator simultaneously was outside the scope of this dissertation. Hence, the results were manually collected after every run of the simulation, with different taxi user data loaded in on each run. Also, since the length of the taxi trajectory varied, the number of service requests made by the taxi users, during a single run was increased. To achieve

this an automated service request process was used that sent requests by taxi users to the cloud server, every 2 minutes. In comparison, [8] experiments on ambulance users that sent service requests every 5 minutes.

Configuration	**Value**
Number of Ants	10
Number of Iterations	100
α	0.002
β	0.0001
Δ	0.001

Table 5.3: ACO Service Placement Algorithm Configuration

The configurations for the ACO based service placement algorithm were reused from previous work. Since these configurations achieved the best results in [8], no additional modifications were made.

5.3.2 Evaluation Metrics

To evaluate the performance of the service placement algorithm, three different metrics have been used. These metrics have been adopted from the work done in [8]. The metrics are described as follows:

- Average Latency
  The average latency is based on the Average Communication Latency between the different edge servers that have been selected for placing the services in. This metric gives an indication of the delay involved with serving a particular user request.

- Average Resource Utilization
  The average resource utilisation metric is based on the equal distribution of load among the selected edge servers. The standard deviation of the load on each selected edge server was used to calculate this metric.

- Average Service Placement Time
  The average service placement time metric is the average time required for a service to be placed in a particular edge server. It is used to understand the time taken by the route prediction algorithm along with the service placement algorithm to place services at edge servers.

The next section will discuss the results obtained using the configuration and the evaluation metrics described in this section.

5.3.3 Service Placement Results and Discussion

The best route prediction algorithms, selected in previous sections, were integrated into the Service Placement Algorithm, to run the experiments discussed in this section. Each simulation run can produce anywhere between 5 to 20 service requests, depending on the length of the Taxi Trajectory being tested. To effectively compare the different route predictions algorithms and the performance of the service placement algorithm, for each run of the simulation, all the results acquired from the many service requests are averaged and then compared. Furthermore, for each algorithm, the simulation is run using five different taxi trajectories. Only five different taxi trajectories were used in our study for evaluation purposes since there was no easier alternative available to effectively run simulations on all five or more taxi trajectories at once. Hence this process was cumbersome and required the collection of results after every run. Finally, to measure the performance improvement, a baseline ACO based service placement model was also compared that was not integrated with a route prediction algorithm.

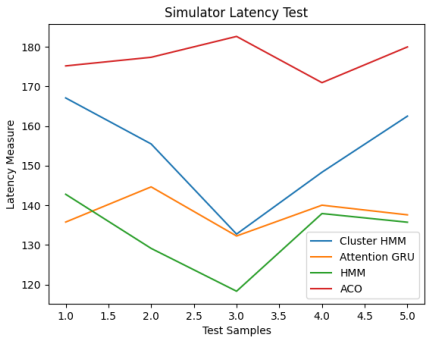

Figure 5.27: Average Latency Per Taxi Trajectory Test

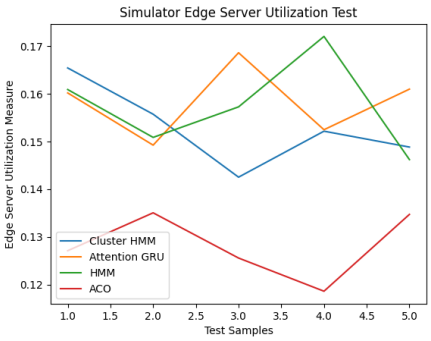

Figure 5.28: Accuracy Utilization Per Taxi Trajectory Test

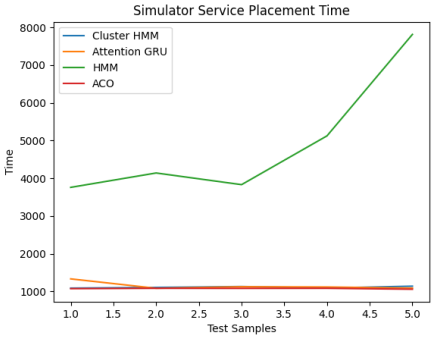

Figure 5.29: Average Service Time Per Taxi Trajectory Test

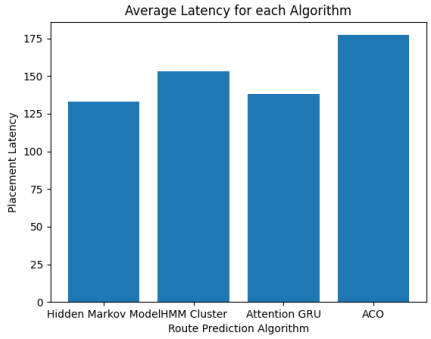

Figure 5.30: Average Latency Per Algorithm

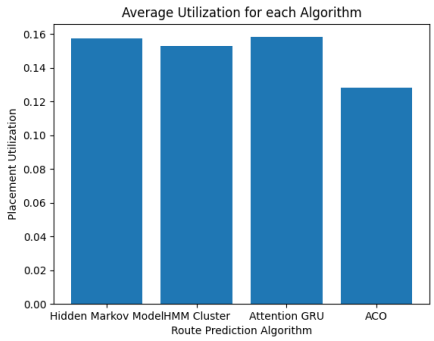

Figure 5.31: Average Utilization Per Algorithm

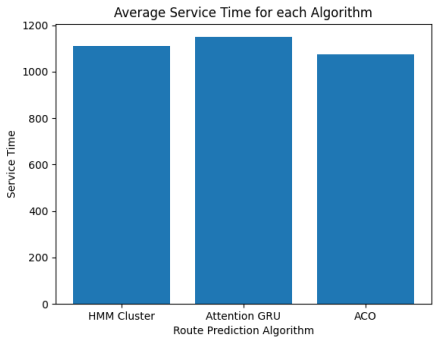

Figure 5.32: Average Service Time Per Algorithm

The average latencies acquired from each of the five different taxi trajectory simulation runs, for each algorithm is compared in Figure 5.27. From this graph, we can observe a big difference in latency between the baseline service placement algorithm, with no route prediction integration, and the service placement algorithms (SPAs) integrated with the selected route prediction models. The graph also shows a difference between the Cluster-Based HMM integrated SPA and the other route prediction integrated SPA

models implemented in this dissertation. It can also be seen, that the Hidden Markov SPA model performs slightly better in this metric. This is in line with the individual performance of the route prediction algorithms discussed in previous sections.

Similarly, the Average Server Utilization for the different algorithms have been compared in Figure 5.28. Again we see a big difference between the performance of the Baseline model and the other algorithms. We can confirm with this, that the route prediction models definitely improves the performance of the service placement algorithm. Similarly, we can also notice a slight improvement in Server Utilization when using a HMM SPA model when compared with the GRU attention SPA model, while the Cluster based HMM SPA Model still has the worst performance, among the three algorithms.

However, in figure 5.29, that compares the average service placement time, the Hidden Markov SPA Model displays the worst the performance. This is because for each future cell prediction, the viterbi algorithm has to be re-run. This, as well as the high number of Grid Cells present, increases the computation time of the algorithm, when compared to the other algorithms. Furthermore, since the Hidden Markov model and the Gru Attention model are run on a separate python server, the service placement time, now includes also the time for making requests via http for route predictions. This is is not the case for the Cluster Based Hidden Markov model as it has been implemented in the Simulator Environment. Also, since the baseline ACO based model doesn't require external route prediction algorithms for placing services on different edge servers, it performs the best on this metric. Therefore, while this metric helps captures the individual route prediction algorithms' computational performance, these algorithms can constantly be optimised and incorporated into the simulator environment for quicker placement time. Hence, more emphasis will be given to the other metrics, the Average Latency and the Average Utilization, for performance indication.

Figures 5.32,5.31 and 5.30, show the results averaged over the 5 different trajectories for each algorithm. As discussed, we see a clear distinction between the baseline performance when compared to the other algorithms. Also, the performance of the HMM SPA model is superior to the Cluster-based SPA Model and is slightly better than the GRU attention SPA Model. However, in figure 5.32, only the Cluster-Based HMM SPA model, the GRU attention SPA Model and the baseline model averages are plotted so that a more clear division between the algorithms can be seen. Here we see that the GRU attention model still performs well despite the need to be called over a network.

5.4 Summary

These results prove that the Service Placement algorithm benefits from future route predictions of the Users' trajectory. Moreover, the route predictions algorithm implemented in this dissertation is superior when compared individually and in the context of the service placement algorithm, to a baseline Cluster based HMM model. Finally, between all the metrics used to compare the service placement algorithm, the GRU Attention model performs consistently well in all of them. Finally, if we ignore the service placement time metric, the HMM shows superior performance.

Chapter 6

Conclusions & Future Work

This chapter summarizes the work done in this dissertation and highlights its main contributions. This chapter is then concluded by discussing future research ideas and directions as well as any limitations of this study.

6.1 Summary

This dissertation emphasized the importance of MEC servers. We saw how MEC systems were capable of proving users with low latency and high bandwidth services as compared to Cloud Computing Servers. It was also highlighted that MEC systems are resource constraint, and require algorithms for distributing the load of computationally intensive applications among different edge servers. The dissertation also discussed the challenges associated with MEC servers, and underlined the importance of user mobility for service placement algorithms.

To implement a better a Service Placement Algorithm, different state of the art route prediction were studied and promising algorithms were selected for further analysis in this dissertation. These route prediction algorithm were implemented and studied individually and in the context of a service placement algorithm. The results between the individual performance and the service placement algorithm were analysed and certain patterns were highlighted.

6.2 Contributions

The objective of this research was to compare and analyze state of the art route predictions applied to service placement models, to acquire insight into potential improvements. Furthermore, this dissertation, also aimed to highlight potential route prediction algorithms

that would further improve the service placement algorithm.

To achieve these objectives, a Hidden Markov Model and variations of a Sequence to Sequence model were implemented and compared individually in section 5.3. This study, highlights the improvement in route prediction performance that the Hidden Markov models and Sequence to Sequence models bring, when compared to more simpler models such as a Cluster Based Hidden Markov Model implemented in existing works. In section 5.3, these algorithm were integrated with a existing Service Placement Algorithm. The results described in section 5.3.3 showed a clear improvement in performance when compared with a baseline service placement algorithm and a Cluster Based HMM SPA model. It was also highlighted that the Hidden Markov SPA Model was capable of performing better in the two main metrics used, but suffered on the Placement Time metric due to the nature of the algorithm. Also, this research underlined the consistent performance of the GRU Attention Based SPA model and highlighted the model's potential for future work.

6.3 Limitations

During the implementation as well as the testing of the various algorithms, many challenges were faced, and a more simplistic approach had to be accepted. The limitations faced in this dissertation have been summarized below:

- Prediction Algorithms
  While the route prediction algorithms used in this dissertation perform well, there is still a lot of scope for improvement. For instance, further optimization to the Hidden Markov Model, to predict future trajectories using only one run of the Viterbi Algorithm could have been studied and implemented.

- Testing
  There were a few places where testing the route prediction algorithm could have been improved. For instance, while the dataset included 1.6 million records, only 10000 records were used to finally train all the models, and only 1000 records were used for testing the model. This was done due to the increasingly long load times of the Simulator, whenever the number of clusters of the Cluster Based Hidden Markov Model was increased. This also led to an inefficient use of the complex GRU Based Attention model, that performs better when trained on bigger datasets.

- Taxi Movement
  Taxi movement in the Simulator suffered, due to the implicit randomization of the

movement in the simulation. To address this, different configurations that were present in the Simulator could have been studied and used. Also, simulation of multiple taxis, wasn't possible due to the nature of the study. That is real taxi trajectories had to be fed into the simulator, as compared to using randomized locations and simulated movement in previous studies. Because of this, the results had to be calculated and collected after every run for each algorithm, which increased the time spent testing the algorithms.

6.4 Future Work

Since this research, was able to prove the ability of route prediction algorithms in improving the performance of Service Placement Algorithms, there appears to be great potential for future research along many directions. Some of those possibilities have been discussed below:

- Due to the complexity of the dissertation by itself, certain algorithms like transformers and Variable Markov Models were not investigated and can be implemented for comparison in future work. For instance, transformer networks have proven to be better at sequence to sequence prediction tasks as compared to the RNN based Sequence to Sequence networks.

- Comparison, between a grid based and edge based map matching was experimented with during the implementation phase of this dissertation. However, problems arose with the number of Edge Links that were being generated, even for the smaller portion of the Porto Map that was finally used. This impacted performance and required more data to train, which wasn't feasible in this study. However, Edge based map matching is the more popular map-matching technique and can be further studied for future route prediction algorithms.

- Finally, a better Service Placement Algorithm could have been studied and implemented. There are many studies in the literature that make use of a more complex meta-heuristic Reinforcement Learning algorithm for service placement. RL algorithms, have surpassed human level intelligence in many applications and is a promising field for future study.

Bibliography

[1] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[2] S. Barbarossa, S. Sardellitti, and P. Di Lorenzo. Communicating while computing: Distributed mobile cloud computing over 5g heterogeneous networks. *IEEE Signal Processing Magazine*, 31(6):45–55, 2014.

[3] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[4] L. Chen, C. Shen, P. Zhou, and J. Xu. Collaborative service placement for edge computing in dense small cell networks. *IEEE Transactions on Mobile Computing*, 2019.

[5] M. Chiang and T. Zhang. Fog and iot: An overview of research opportunities. *IEEE Internet of things journal*, 3(6):854–864, 2016.

[6] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[7] N. R. Chopde and M. Nichat. Landmark based shortest path detection by using a* and haversine formula. *International Journal of Innovative Research in Computer and Communication Engineering*, 1(2):298–302, 2013.

[8] C. Christian, S. Sergej, P. Andrei, K. Aqeel, and C. Siobhán. Maaco a dynamic service placement model for smart cities. 2014.

[9] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[10] Y. Dai, Y. Ma, Q. Wang, Y. L. Murphey, S. Qiu, J. Kristinsson, J. Meyer, F. Tseng, and T. Feldkamp. Dynamic prediction of drivers' personal routes through machine

learning. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8. IEEE, 2016.

[11] H. T. Dinh, C. Lee, D. Niyato, and P. Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless communications and mobile computing*, 13(18):1587–1611, 2013.

[12] M. Dorigo and T. Stützle. The ant colony optimization metaheuristic: Algorithms, applications, and advances. In *Handbook of metaheuristics*, pages 250–285. Springer, 2003.

[13] P. Ebel, I. E. Göl, C. Lingenfelder, and A. Vogelsang. Destination prediction based on partial trajectory data. In *2020 IEEE Intelligent Vehicles Symposium (IV)*, pages 1149–1155. IEEE, 2020.

[14] Ericson. Ericsson mobility report. 2016. URL `https://www.ericsson.com/assets/local/mobility-report/documents/2016/Ericsson-mobility-report-june-2016.pdf`.

[15] N. Fernando, S. W. Loke, and W. Rahayu. Mobile cloud computing: A survey. *Future generation computer systems*, 29(1):84–106, 2013.

[16] A. Filali, A. Abouaomar, S. Cherkaoui, A. Kobbane, and M. Guizani. Multi-access edge computing: A survey. *IEEE Access*, 8:197017–197046, 2020.

[17] L. Irio, A. Ip, R. Oliveira, and M. Luís. An adaptive learning-based approach for vehicle mobility prediction. *IEEE Access*, 9:13671–13682, 2021.

[18] D. Jurafsky and J. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, volume 2. 02 2008.

[19] Y. Lassoued, J. Monteil, Y. Gu, G. Russo, R. Shorten, and M. Mevissen. A hidden markov model for route and destination prediction. In *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–6. IEEE, 2017.

[20] S. Lee, S. Lee, and M.-K. Shin. Low cost mec server placement and association in 5g networks. In *2019 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 879–882. IEEE, 2019.

[21] F. Li, Q. Li, Z. Li, Z. Huang, X. Chang, and J. Xia. A personal location prediction method based on individual trajectory and group trajectory. *IEEE Access*, 7:92850–92860, 2019.

[22] J. Li, W. Liang, M. Chen, and Z. Xu. Mobility-aware dynamic service placement in d2d-assisted mec environments. In *2021 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6. IEEE, 2021.

[23] W. Y. B. Lim, N. C. Luong, D. T. Hoang, Y. Jiao, Y.-C. Liang, Q. Yang, D. Niyato, and C. Miao. Federated learning in mobile edge networks: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 22(3):2031–2063, 2020.

[24] Z. C. Lipton, J. Berkowitz, and C. Elkan. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*, 2015.

[25] W. Liu and Y. Shoji. Deepvm: Rnn-based vehicle mobility prediction to support intelligent vehicle applications. *IEEE Transactions on Industrial Informatics*, 16(6): 3997–4006, 2019.

[26] Y. Ma, W. Liang, J. Li, X. Jia, and S. Guo. Mobility-aware and delay-sensitive service provisioning in mobile edge-cloud networks. *IEEE Transactions on Mobile Computing*, 2020.

[27] F. Meneghello, D. Cecchinato, and M. Rossi. Mobility prediction via sequential learning for 5g mobile networks. In *2020 16th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)(50308)*, pages 1–6. IEEE, 2020.

[28] L. Moreira-Matias, J. Gama, M. Ferreira, J. Mendes-Moreira, and L. Damas. Predicting taxi–passenger demand using streaming data. *IEEE Transactions on Intelligent Transportation Systems*, 14(3):1393–1402, 2013.

[29] A. Moubayed, A. Shami, P. Heidari, A. Larabi, and R. Brunner. Edge-enabled v2x service placement for intelligent transportation systems. *IEEE Transactions on Mobile Computing*, 20(4):1380–1392, 2020.

[30] F. D. N. Neto, C. de Souza Baptista, and C. E. Campelo. Combining markov model and prediction by partial matching compression technique for route and destination prediction. *Knowledge-Based Systems*, 154:81–92, 2018.

[31] S. O'Dea. Forecast number of mobile users worldwide 2020-2025. 2021. URL `https://www.statista.com/statistics/218984/number-of-global-mobile-users-since-2010/`.

[32] T. Ouyang, Z. Zhou, and X. Chen. Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing. *IEEE Journal on Selected Areas in Communications*, 36(10):2333–2345, 2018.

[33] Q.-V. Pham, F. Fang, V. N. Ha, M. J. Piran, M. Le, L. B. Le, W.-J. Hwang, and Z. Ding. A survey of multi-access edge computing in 5g and beyond: Fundamentals, technology integration, and state-of-the-art. *IEEE Access*, 8:116974–117017, 2020.

[34] P. Porambage, J. Okwuibe, M. Liyanage, M. Ylianttila, and T. Taleb. Survey on multi-access edge computing for internet of things realization. *IEEE Communications Surveys & Tutorials*, 20(4):2961–2991, 2018.

[35] Pytorch. Categorical cross entropy loss. 2019. URL `https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html`.

[36] L. Rabiner and B. Juang. An introduction to hidden markov models. *ieee assp magazine*, 3(1):4–16, 1986.

[37] P. Rathore, D. Kumar, S. Rajasegarar, M. Palaniswami, and J. C. Bezdek. A scalable framework for trajectory prediction. *arXiv preprint arXiv:1806.03582*, 2018.

[38] B. Richerzhagen, D. Stingl, J. Ruckert, R. Steinmetz, et al. Simonstrator: Simulation and prototyping platform for distributed mobile applications. In *The 8th EAI International Conference on Simulation Tools and Techniques (ACM SIMUTOOLS 2015)*, pages 99–108, 2015.

[39] D. Sabella, A. Vaillant, P. Kuure, U. Rauschenbach, and F. Giust. Mobile-edge computing architecture: The role of mec in the internet of things. *IEEE Consumer Electronics Magazine*, 5(4):84–91, 2016.

[40] D. Sabella, V. Sukhomlinov, L. Trang, S. Kekki, P. Paglierani, R. Rossbach, X. Li, Y. Fang, D. Druta, F. Giust, et al. Developing software for multi-access edge computing. *ETSI white paper*, 20:1–38, 2019.

[41] M. Satyanarayanan. Fundamental challenges in mobile computing. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 1–7, 1996.

[42] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4):14–23, 2009.

[43] P. Schulz, M. Matthe, H. Klessig, M. Simsek, G. Fettweis, J. Ansari, S. A. Ashraf, B. Almeroth, J. Voigt, I. Riedel, et al. Latency critical iot applications in 5g: Perspective on the design of radio interface and network architecture. *IEEE Communications Magazine*, 55(2):70–78, 2017.

[44] S. Shahzadi, M. Iqbal, T. Dagiuklas, and Z. U. Qayyum. Multi-access edge computing: open issues, challenges and future perspectives. *Journal of Cloud Computing*, 6 (1):1–13, 2017.

[45] A. Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, 2020.

[46] R. Simmons, B. Browning, Y. Zhang, and V. Sadekar. Learning to predict driver route and destination intent. In *2006 IEEE Intelligent Transportation Systems Conference*, pages 127–132. IEEE, 2006.

[47] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[48] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella. On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration. *IEEE Communications Surveys & Tutorials*, 19(3):1657–1681, 2017.

[49] D. L. Urban and D. O. Wallin. Introduction to markov models. In *Learning landscape ecology*, pages 129–142. Springer, 2017.

[50] C. Wang, L. Ma, R. Li, T. S. Durrani, and H. Zhang. Exploring trajectory prediction through machine learning methods. *IEEE Access*, 7:101441–101452, 2019.

[51] G. J. Woeginger. Exact algorithms for np-hard problems: A survey. In *Combinatorial optimization—eureka, you shrink!*, pages 185–207. Springer, 2003.

[52] H. Wu, S. Deng, W. Li, J. Yin, X. Li, Z. Feng, and A. Y. Zomaya. Mobility-aware service selection in mobile edge computing systems. In *2019 IEEE International Conference on Web Services (ICWS)*, pages 201–208. IEEE, 2019.

[53] G. Xue, Z. Li, H. Zhu, and Y. Liu. Traffic-known urban vehicular route prediction based on partial mobility patterns. In *2009 15th International Conference on Parallel and Distributed Systems*, pages 369–375. IEEE, 2009.

[54] Y. Yu, X. Si, C. Hu, and J. Zhang. A review of recurrent neural networks: Lstm cells and network architectures. *Neural computation*, 31(7):1235–1270, 2019.

[55] Y. Zhai, T. Bao, L. Zhu, M. Shen, X. Du, and M. Guizani. Toward reinforcement-learning-based service deployment of 5g mobile edge computing with request-aware scheduling. *IEEE Wireless Communications*, 27(1):84–91, 2020.

[56] D. Zhang, B.-H. Yan, Z. Feng, C. Zhang, and Y.-X. Wang. Container oriented job scheduling using linear programming model. In *2017 3rd International Conference on Information Management (ICIM)*, pages 174–180. IEEE, 2017.

[57] L. Zhao, J. Liu, Y. Shi, W. Sun, and H. Guo. Optimal placement of virtual machines in mobile edge computing. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pages 1–6. IEEE, 2017.

[58] X. Zhao, Y. Shi, and S. Chen. Maesp: Mobility aware edge service placement in mobile edge networks. *Computer Networks*, 182:107435, 2020.

[59] L. Zuo, L. Shu, S. Dong, C. Zhu, and T. Hara. A multi-objective optimization scheduling method based on the ant colony algorithm in cloud computing. *Ieee Access*, 3:2687–2699, 2015.

Appendix

.1 Code for Route Prediction Algorithms

.1.1 Attention Based Sequence to Sequence Network

```
class EncoderNetwork(nn.Module):
    def __init__(self, inp_size, embed_size, hidden_size,
                 n_layers=1, dropout=0.5):
        super(EncoderNetwork, self).__init__()
        self.inp_size = inp_size  #400
        self.hidden_size = hidden_size  # 512
        self.embed_size = embed_size  # 256
        self.embeddingLayer = nn.Embedding(input_size, embed_size)
        self.lstm = nn.LSTM(embed_size, hidden_size, n_layers,
                            dropout=dropout, bidirectional=True)

    def forward(self, src, hidden=None):
        # src shape => (6, 128)
        embedded = self.embed(src)

        outputs, (hidden, cell) = self.lstm(embedded, hidden)

        hidden = (hidden[0:1] + hidden[1:2])
        cell = (cell[0:1] + cell[1:2])

        # Add forward and backward hidden states of the Bidirectional LSTM
        # Outputs => (Num_layers, 128, 512 (Hidden Size))
        outputs = (outputs[:, :, :self.hidden_size] +
                   outputs[:, :, self.hidden_size:])
        return outputs, hidden, cell


class AttentionLayer(nn.Module):
    def __init__(self, hidden_size):
        super(AttentionLayer, self).__init__()
```

```
        self.hidden_size = hidden_size
        self.mlp_atention = nn.Linear(self.hidden_size * 2, 1)

    def forward(self, decoder_hidden, enc_hidden_states):
        timestep = encoder_outputs.size(0)
        hidden = hidden.squeeze()
        # Repeat the Previous Decoder Hidden States to match Encoder Hidden States
        h = decoder_hidden.repeat(timestep, 1, 1).transpose(0, 1)
        alignment_scores = F.tanh(self.attn(torch.cat([decoder_hidden, enc_hidden_states], 2)))
        return F.softmax(alignment_scores)


class DecoderNetwork(nn.Module):
    def __init__(self, embed_size, hidden_size, pred_size,
                 n_layers=1, dropout=0.2):
        super(DecoderNetwork, self).__init__()
        self.embed_size = embed_size  # 400
        self.hidden_size = hidden_size  # 512
        self.pred_size = pred_size  #100
        self.n_layers = n_layers  # 1

        self.embedding_layer = nn.Embedding(pred_size, embed_size)
        self.dropout_layer = nn.Dropout(dropout, inplace=True)
        self.attention_net = AttentionNetwork(hidden_size)
        self.lstm = nn.LSTM(hidden_size + embed_size, hidden_size,
                            n_layers, dropout=dropout)
        self.output_layer = nn.Linear(hidden_size * 2, pred_size)

    def forward(self, input,enc_hidden, previous_hidden_state, previous_cell_state):
        # Get the embedding of the current input word (last output word)
        embedded = self.embedding_layer(input)

        # Calculate attention weights and apply to encoder hidden states to generate the context vector
        attn_weights = self.attention_net(previous_hidden_state, enc_hidden)
        context = attn_weights.bmm(enc_hidden.transpose(0, 1))
        rnn_input = torch.cat([embedded, context], 2)
        output, (hidden,cell) = self.lstm(rnn_input, [previous_hidden_state, previous_cell_state])
        output = self.output_layer(torch.cat([output, context], 1))
```

```
        output = F.log_softmax(output, dim=1)
        return output, hidden,cell, attn_weights

class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder):
        super(Seq2Seq, self).__init__()
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, source, target, teacher_forcing_ratio=0.5):
        # Get the batch size
        batch_size = source.size(1)

        # Get the Length of the Target Vector - Can feed in a dummy tensor as well
        len = target.size(0)

        # Get Vocab Size
        vocab_size = self.decoder.output_size

        # Initiate a vector of Zeros for output values
        output_tensor = torch.zeros(len, batch_size, vocab_size)

        encoder_hidden_states, hidden_state, cell_state = self.encoder(source)

        # Get the First value of the output
        decoder_inp = Variable(target[0])  # 1

        # Starting with one as the first value has already been taken
        for t in range(1, len):
            # Calling Decoder with Output shape => (1, 128)
            # Encoder Hidden States size => (Num_layers * 2, 128, 512)
            output, hidden,cell, attn_weights = self.decoder(
                decoder_inp, hidden_state,cell_state, encoder_hidden_states)
            if random.random() < teacher_forcing_ratio:
                decoder_inp = Variable(trg.data[t] ).cuda()
            else:
                output_tensor[t] = np.argmax(output)
                decoder_inp =  np.argmax(output)


        return output_tensor
```

.1.2 Hidden Markov Model

```
class HMM():
    def __init__(self, routes, N, obs_states_n):
        self.routes = routes
        self.N = N
        self.transitionProb = []
        self.emissionProb = []
        self.intialProb = []
        self.sequence_freq = []
        self.obs_states_n = obs_states_n
        self.id_to_seq = {}
        _ = self.get_transition_prob()
        _ = self.get_initial_prob()
        _ = self.get_emission_prob()


    def get_transition_prob(self):
        seq_to_id_count = 0
        n = self.N
        self.seq_to_id = {}
        for route in self.routes:

            # For loop to map the route to sequences and calculate the
    trans freq and the sequence freq
            for i in range(0, len(route)-n+1): # (0 to ( 10-2) = 8) i.e (0
    to (7 +2)  = 9) - 0 to including 9 is 10 places

                sequence = tuple(route[i:i+n])
                if  sequence not in self.seq_to_id.keys():
                    self.seq_to_id[sequence] = seq_to_id_count
                    self.id_to_seq[seq_to_id_count] = sequence
                    seq_to_id_count = seq_to_id_count +1




        self.tempTransitionFreq  =  np.zeros((len(self.seq_to_id), len(self
    .seq_to_id)))
        self.sequence_freq = np.zeros((len(self.seq_to_id),))
```

```
36                      # Mapping storing the sequence in the sequence freq
    table


38
        # Calculation of the trans freq
40      for route in self.routes:
            for i in range(0, len(route)-n+1):
42
                # Hidden States Frequence - Sequence Freq Calculation
44              sequence = self.seq_to_id[tuple(route[i:i+n])]

46              count = self.sequence_freq[sequence] +1
                self.sequence_freq[sequence] = count
48
                if not i>=(len(route)-n):
50                  sequence1 = self.seq_to_id[tuple(route[i:i+n])]
                    sequence2 = self.seq_to_id[tuple(route[i+1:i+n+1])]
52
                    count = self.tempTransitionFreq[sequence1,sequence2] +1
54                  self.tempTransitionFreq[sequence1,sequence2] = count


56
        self.transitionProb = np.zeros((len(self.seq_to_id), len(self.
    seq_to_id)))
58
        # Transition Probability Calculation Using Transition Frequenc
60      for i in range(0, len(self.transitionProb)):
            current_sum = self.tempTransitionFreq[i,:].sum()
62          if current_sum > 0:
                self.transitionProb[i,:] = [ float("{:.5f}".format(trans/
    current_sum)) for trans in self.tempTransitionFreq[i,:]]
64              if self.transitionProb[i,:].sum() > 1.001:
                    print("The row sum was bigger than 1 at row id: ", i)
66
        return self.transitionProb
68
    def get_initial_prob(self):
70      self.intialProb = np.zeros((len(self.seq_to_id),))
        # calculate total sum of frequencies for all the hidden states
72      total_sum = self.sequence_freq.sum()
        # calculate the initial Probability of all the hidden states by
    dividing by the total sum
74      self.intialProb = self.sequence_freq/total_sum
        for i in self.intialProb:
```

```
76              if i > 1.0:
                    print("Bigger than 1 at id: ", i)
78          return self.intialProb

80      def get_emission_prob(self):
            self.emissionProb = np.zeros((len(self.seq_to_id), self.
    obs_states_n))
82          # For all the hidden states loop
            for sequence in self.seq_to_id:
84              seq_id = self.seq_to_id[sequence]

86              # get length of current sequence
                total_count = len(sequence)
88              temp_cell_count = {}

90              # Calculate frequency of the Hidden States inside the current
    sequence
                for cell in sequence:
92                  if cell not in temp_cell_count:
                        temp_cell_count[cell] = 1
94                  else:
                        new_cell_count = temp_cell_count[cell] + 1
96                      temp_cell_count[cell] = new_cell_count

98              # Divide freq of each hidden state in sequence by total length
    of sequence.
                for cell in temp_cell_count.keys():
100                 self.emissionProb[seq_id, cell] = temp_cell_count[cell]/
    total_count

102             if self.emissionProb[seq_id].sum() > 1:
                    print("Greater than one at id: ", seq_id)
104
            return self.emissionProb
106
        def viterbi(self, O):
108
            transitionProb = self.transitionProb
110         intialProb = self.intialProb
            emissionProb = self.emissionProb
112
            n_hstates = transitionProb.shape[0]     # Number of states
114         num_obs = len(O)  # Length of observation sequence
```

```
        # Initialize Viterbi and Backtracking zero matrices
        viterbi = np.zeros((n_hstates, num_obs+1))
        back_track = np.zeros((n_hstates, num_obs)).astype(np.int32)
        viterbi[:, 0] = np.multiply(intialProb, emissionProb[:, O[0]])

        # Compute Viteribi and Backtracking Matrices in a loop
        for n in range(1, num_obs +1):
            for i in range(n_hstates):
                if n == num_obs:
                    temp_product = np.multiply(transitionProb[:, i],
    viterbi[:, n-1])
                    viterbi[i, n] = np.max(temp_product)
                    back_track[i, n-1] = np.argmax(temp_product)
                else:
                    temp_product = np.multiply(transitionProb[:, i],
    viterbi[:, n-1])
                    viterbi[i, n] = np.max(temp_product) * emissionProb[i,
    O[n]]
                    back_track[i, n-1] = np.argmax(temp_product)



        # Backtracking
        ouputsequence = np.zeros(num_obs+1).astype(np.int32)
        ouputsequence[-1] = np.argmax(viterbi[:, -1])
        for n in range(num_obs -1, -1, -1):
            ouputsequence[n] = back_track[int(ouputsequence[n+1]), n]

        seq = []
        for sequence in ouputsequence:
            seq.append(self.id_to_seq[int(sequence)])
        return seq, viterbi, back_track
```