LEARNING DRIVER PREFERENCES FOR FREEWAY MERGING

USING MULTITASK IRL

by

SANATH BHAT

(Under the Direction of Prashant Doshi)

ABSTRACT

Most automobile manufacturers today have invested heavily in the research and design of implementing autonomy in their cars. One important and challenging problem faced by a self-driven car on highways is merging into the highway from an acceleration ramp. Successful merging needs consideration of the behaviors of cars driving in the outermost highway lane which is adjacent to the merging lane, especially, the behaviors of those cars that would potentially become the leading or following car after a successful merge. We attempt to predict the motivation for the behaviors of those cars driving on the outermost highway lanes near the merging area hypothesizing that they perform a series of tasks each of which is driven by different motivations while passing through each section of the merging area. We use a Hierarchical Bayesian model to model the preferences in each task and the priors over those preferences.

INDEX WORDS: Inverse Reinforcement Learning, Hierarchical Bayesian Model, Multitask, Highway Merging, NGSIM, Likelihood Weighting

LEARNING DRIVER PREFERENCES FOR FREEWAY MERGING USING MULTITASK IRL

by

SANATH BHAT

B.E., University of Mumbai, India, 2011

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2017

LEARNING DRIVER PREFERENCES FOR FREEWAY MERGING

USING MULTITASK IRL

by

SANATH BHAT

Major Professor:	Prashant Doshi
Committee:	Suchendra Bhandarkar
	Gauri Datta

Electronic Version Approved:

Suzanne Barbour
Dean of the Graduate School
The University of Georgia
August 2017

DEDICATION

To Mom, Dad and Bharath

ACKNOWLEDGEMENTS

I'd like to thank my professor Dr. Prashant Doshi for his constant feedback and guidance throughout the duration of my research. I'd also like to thank all my Thinc Lab colleagues who have provided valuable feedback during our weekly meetings. Also, special thanks to Christos Dimitrakakis of the University of Lille for his prompt and valuable explanations about his paper despite his busy schedule.

Finally, I'd like to thank my family for their constant support both morally and financially without whom I couldn't possibly come this far from home and fulfil my dream of obtaining a Master's degree from a prestigious institution in the US.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

An autonomous car is one capable of perceiving its environment through its sensors and taking appropriate navigation actions based on the situation it finds itself in. Typical sensors including but not limited to radar, lidar, GPS and cameras provide the sensory information to advanced control systems which interpret the information and take appropriate decisions to navigate the car along its path to its destination. Although the first autonomous cars appeared in the 1980s, the first major milestone was achieved by Carnegie Mellon University's NavLab 5[1] in 1995 by completing a coast-to-coast drive of nearly 3000 miles with 98.2% autonomy. Since then, several major car manufacturers have invested heavily into the research of vehicle autonomy. While the problem of autonomous decision making for cars can be broken down into several smaller problems like pathing, obstacle avoidance, passing other vehicles, parking in appropriate places, following relevant signage, just to name a few, this work of research primarily deals with the problem of merging into a highway from an acceleration ramp.

1.1 Problem Description and Motivation

Highway lane merging is a challenging task for an autonomous car since we often see drivers changing their general driving style near entrance ramps causing the autonomous car to predict this change in style and adapt to any situation changes accordingly. Possible reasons for change in driving style may be (i)extra cautiousness or courteousness due to which the driver breaks and allows a merging car to merge in front even if he can clearly pass the merging car and

get ahead of it or (ii)haste or riskiness due to which the driver accelerates and makes a risky attempt to pass the merging car and stay ahead of it. These deviating attitudes of drivers are often anticipated and correctly adjusted to by a human driver who is attempting to merge into the highway. But for an autonomous car this becomes an added challenge because the autonomous system expects only the behavior of the car that it sees while it is on the ramp when approaching the merging area and does not expect any deviation from that behavior when merging. Thus, we need a system that can preemptively expect a change in the policy of the highway driver. We shall define our problem and naming conventions of different involved cars using Figure 1.1 shown below.

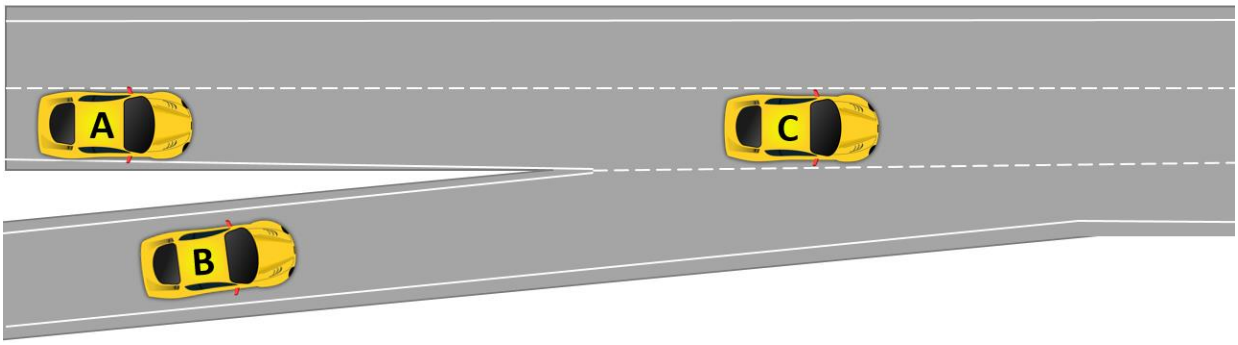

Figure 1.1 Highway Lane Merging Problem Conventions

In Figure 1.1, car B is the autonomous car trying to merge into the outermost highway lane on which car A is the first car behind car B and car C is the first car in front of car B. It would seem that car B would eventually merge into the highway between cars A and C but this is not always the case. Car A sometimes demonstrates risky behavior[2] and accelerates and gets ahead of car B causing car B to slow down and merge behind it. This acceleration is unusual to the driving style of car A when not near merging zones. Another situation arises when car A can clearly maintain

its current speed and still pass car B but instead brakes and allows car B to merge in front of it. Our objective is to identify the varying motivations of car A while passing through distinct phases of merging. Obviously, our first problem would be to establish the existence and distinction of the separate phases. Without loss of generality, we shall refer to all the three vehicles as 'cars' although this need not be the case always.

1.2 Background

This work mainly deals with probabilistic decision making and planning and makes use of statistical techniques of estimation. In this section, we shall discuss fundamental concepts related to decision making theory as well as some statistical concepts and techniques that we use in this thesis.

1.2.1 Markov Decision Processes and related concepts

Many real world stochastic decision making problems cannot be effectively solved with 'one-shot' or episodic planning in which an agent's actions are planned only considering the current situation and utility of the immediate action while disregarding the utility of probable future situations. However, proper balancing of risk and reward is effectively achieved by sequential planning wherein possible futures emanating from an agent's action are considered and the sequence of decisions dictates the agent's utility. Because of the uncertainty of action outcomes, such stochastic environments are assumed to have the Markovian property meaning the outcome of an action depends only on the current state and not the sequence of past states of the agent. The Markov Decision Process (MDP) framework provides a well-established mathematical formulation for discrete time stochastic processes.

A MDP is a 5-tuple {S, A, T, R, γ} where

- S is the set of states of the agent
- A is the set of actions available to the agent
- T(s, a, s'): S x A x S → [0, 1] is the transition function containing probabilities P(s' | s, a) which is the probability of reaching state s' when action a is taken in state s
- R(s, a): S x A → $\mathbb{R}$ is the reward function representing the reward of doing action *a* in state *s*
- γ ϵ [0,1] is the discount factor representing the effect of future rewards on the agent's utility

The main objective of a MDP problem is to find a policy π which specifies the action π(*s*) to perform when the agent is in state *s*. This kind of policy, π(*s*) : S → A, is called a deterministic policy wherein the action to be taken in state s is determined to be *a*. There are also stochastic policies π(s, a) : S x A → [0, 1] which represent a probability distribution P(A| *s*) over all actions for state *s*. In simulation of such policies, an action is chosen arbitrarily in state s based on the probability P(a|s). To compute such a policy, we introduce the concept of Value functions V(s) representing the utility of each state. The Value function is defined in [3] as the sum of the immediate reward of performing action *a* in a state *s* and the expected discounted utility of the next state assuming *a* is the optimal action chosen in state *s*. It is given by the Bellman equation

$$V(s) = \max_{a \epsilon A} \left(R(s,a) + \gamma \sum_{s'} T(s,a,s') * V(s') \right) \quad \text{... Eq 1.1}$$

This is a non-linear system of equations owing to the max operator and hence linear algebra cannot be applied to solve these. The standard approach to calculate the Value function is using well-known iterative methods[3] like Value Iteration and Policy Iteration. We shall describe only the Value Iteration algorithm here as we have used it in our algorithm.

Value iteration is an iterative process involving multiple Bellman update steps until convergence of the Value function V(s). The Value Iteration algorithm[3] is shown in Figure 1.2. V(s) is initialized randomly or simply as V(s)=0, $\forall s \epsilon S$. We then perform a Bellman update given by

$$V_{i+1}(s) = \max_{a \epsilon A} \left(R(s,a) + \gamma \sum_{s'} T(s,a,s') * V_i(s') \right) \quad \text{... Eq 1.2}$$

The subscript *i* refers to the time-step/iteration. This iterative update for each state continues until $V_{i+1}(s)$ converges i.e. until $\forall s$, $|V_{i+1}(s) - V_i(s)|$ falls below acceptable error which is given by $\varepsilon(1-\gamma)/\gamma$. ε is chosen to be a small value as per desired accuracy.

function VALUE-ITERATION(mdp, ϵ)
returns a utility function
inputs: mdp, an MDP with states S, actions A, transition model T(s,a,s'),
rewards R(s,a), discount γ
ϵ, the maximum error allowed in the utility of any state
local variables: V, V', vectors of utilities for states in S, initially zero
δ, the maximum change in the utility of any state in an iteration
repeat
$V \leftarrow V'$; $\delta \leftarrow 0$
for each state s in S do
$V'[s] \leftarrow \max_{a \in A}(R(s,a) + \gamma \sum_{s'} T(s,a,s') * V[s'])$
if $|V'[s] - V[s]| > \delta$ then $\delta \leftarrow |V'[s] - V[s]|$
until $\delta < \epsilon(1-\gamma)/\gamma$
return V

Figure 1.2 Value Iteration algorithm

Once V(s) is computed, the optimal deterministic policy $\pi^*(s)$ with respect to R(s, a) is computed as the action which maximizes the expected utility of subsequent state.

$$\pi^*(s) = \underset{a \epsilon A}{\operatorname{argmax}} \sum_{s'} T\,(s, a, s') * V(s') \qquad \text{... Eq 1.3}$$

Another important concept related to MDPs is the Q-function Q(s, a) which gives value of doing action a in state s. The Q-function can be easily calculated from the Value function as

$$Q(s, a) = R(s, a) + \gamma \sum_{s'} T\,(s, a, s') * V(s') \qquad \text{... Eq 1.4}$$

1.2.2 Likelihood weighting[3]

Exact inferencing in Bayesian networks becomes intractable as the number of nodes increases and/or if the networks are 'multiply connected' meaning there are more than one paths between some nodes in the network. In such large networks, approximate inferencing is often the most viable and practical solution. One of the simplest and most robust classes of approximate inferencing algorithms is Monte Carlo algorithms. These algorithms use randomized sampling which involves generating samples from prior distributions and arriving at a posterior distribution and the accuracy of the calculated posterior increases with the number of samples generated. There are two main types of sampling algorithms – direct sampling and Markov chain sampling.

The most basic form of direct sampling is Rejection sampling which involves generating samples from a prior distribution and then rejecting those samples that do not support the evidence. It is one of the simplest type of Monte Carlo methods which can be used to compute conditional probabilities of query variables(X) given the values e of evidence variables E. However, rejection sampling does not perform well when the number of evidence variables is large or the evidence is rarely occurring simply because matching of the all evidence variables becomes a rare occurrence and thus most samples get thrown away.

Instead of generating and rejecting samples that do not support the evidence, likelihood weighting circumvents the wastage of samples by only generating samples that are consistent with the evidence in the first place. When drawing samples, likelihood weighting first fixes the evidence variables to their value (E = e) and then for the remaining variables it generates samples from the prior of those variables which is(are) the parent node(s) of that variable. However, all the samples are not equal. Each sample is weighted proportionately to how well it supports the evidence. Thus, each sample is associated with a weight corresponding to its likelihood and the frequency of each sample corresponds to its prior probability. However, it must be noted that the same sample can have different weights during sampling. This is because the part of the sample used to determine similarity to a previous sample is only the set of query variables X and so two samples with same values for the query variables but different values for non-query variables are still considered the same. In such cases, the weights of that sample are added. Consider the Bayesian network in Fig 1.3.

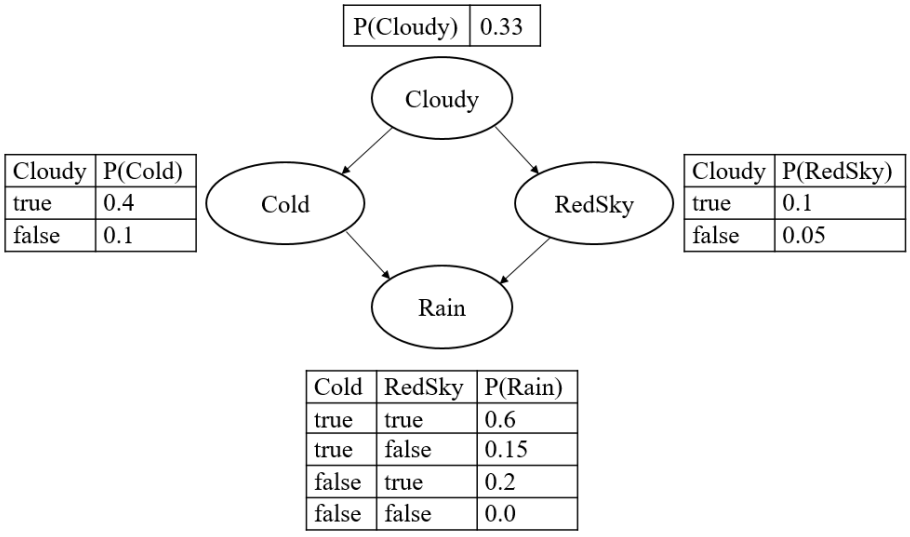

Figure 1.3 Sample Bayesian network with 4 nodes

Suppose using the network in Figure 1.3, we wish to compute the conditional probability P(RedSky | Cloudy = true, Rain = false). We initialize the weight w of the sample to 1. During the sample generation, for each variable X_i:

i. If X_i is an evidence variable with value x_i in e, then multiply the weight by $P(X_i|Parents(X_i))$

ii. Else sample X_i from $P(X_i|Parents(X_i))$

Using these two rules, following are the steps involved in generating one weighted sample when calculating the conditional probability P(RedSky | Cloudy = true, Rain = false).

i. Weight(w) of current sample is initialized to w = 1.

ii. Cloudy is an evidence variable with value Cloudy=true. Therefore, we update w as

$$w = w \times P(Cloudy = true) = 0.33$$

iii. Cold is not an evidence variable. Therefore, we sample its value from its conditional distribution given its parent i.e. P(Cold | Cloudy = true). Suppose we get true.

iv. RedSky is not an evidence variable. Therefore, we sample its value from its conditional distribution given its parent i.e. P(RedSky | Cloudy = true). Suppose we get false.

v. Finally, Rain is an evidence variable with value Rain=true. Therefore, we update w as

$$w = w \times P(Rain = false \mid Cold = true, RedSky = false) = 0.33 \times 0.85 = 0.2805$$

This sample is counted under RedSky = false, this being the only query variable and added to the previous weight, if any, of RedSky=false.

We shall now show how the estimates of likelihood weighting are true approximations. Let w(z, e) be the weight of a sample consisting of non-evidence variables in Z and evidence variables in E. The weight is given by the product of the likelihoods of each evidence variable given its respective parents i.e.,

$$w(z, e) = \prod_i P\left(e_i \middle| Parents(E_i)\right) \quad \text{... Eq. 1.4}$$

The sampling process for the non-evidence variables in Z gives us the sampling probability S(z,e) as

$$S(z, e) = \prod_i P\left(z_i \middle| \text{Parents}(Z_i)\right) \quad \text{... Eq. 1.5}$$

Note that even though evidence variables are considered among parents of Z_i, the evidence among non-parents of Z_i are not considered. Thus, depending on the order of sampling or position of the evidence nodes in the network, samples with zero or almost zero weights may be generated.

Multiplying Eq. 1.4 and Eq. 1.5 gives,

$$S(z, e)\, w(z, e) = \prod_i P\left(z_i \middle| \text{Parents}(Z_i)\right) \prod_i P\left(e_i \middle| \text{Parents}(E_i)\right)$$

$$= P(z, e) \quad \text{…Eq. 1.6}$$

Now for any $x \in X$ which is the query variable, the posterior probability is,

$$\widehat{P}(x \mid e) = \alpha \sum_y N(x, y, e) * w(x, y, e) \qquad \text{By definition of likelihood weighting}$$

$$= \alpha' \sum_y S(x, y, e) * w(x, y, e) \qquad \text{as } N \to \infty$$

$$= \alpha' \sum_y P(x, y, e) \qquad \text{From Eq. 1.6}$$

$$= \alpha' P(x, e) \quad = P(x|e)$$

Likelihood weighting fares poorly as the number of evidence variables increases and the problem exacerbates if the variables occur late in the network. This is seen in most samples having very small weights while a few samples accord a large likelihood to the evidence causing spiked distributions and not representing a true posterior. The number of samples required to represent a decent posterior distribution becomes impractically huge in such cases.

1.2.3 Hierarchical Bayesian Modeling

In Bayesian statistics, the general idea is to use our belief about the probability of the variables being estimated and then updating our belief based on the evidence available. One way of accomplishing this is by approximating the target distribution with a parameterized prior distribution which is easier to sample from and then performing sampling to compute the required approximate marginal over the joint probability distribution. However, sometimes we also have information that leads us to a belief about the parameters of the prior as well. In such cases, instead of choosing fixed prior parameters, it is better to utilize this new level of information and add another layer of uncertainty in the form of a prior over the prior's parameters, i.e. a 'hyperprior'. Such a hierarchical network helps us in getting a better understanding of multiparameter problems and allows us to develop computational strategies to find better solutions to such problems.

A simple hierarchical Bayesian network is described as follows. For simplicity, we shall specify three levels of hierarchy. Suppose a coin with probability p of getting heads is tossed n=100 times. We know that the number of heads obtained in the n trials follows a binomial distribution. Let X denote the number of heads obtained. Therefore, we have,

$X \sim Bin(n, p) = Bin(100, p)$…(Read as 'X is sampled from the distribution Bin(100, p)')

Thus, Bin(100, p) is the prior distribution over X. For an unbiased coin, p = 0.5. But we shall assume that our coin has an unknown bias p which we need to estimate. Now, treating p as another random variable, we shall choose a prior over p which is the hyperprior over X. Let this hyperprior be a Beta distribution, Beta(α, β), that is a well-known probability measure over another probability measure. By fixing α and β (say α=2, β=5), we end the hierarchy at this level. However, if we have additional information about the uncertainty of α and β, we can add another level by adding priors over the α and β hyperparameters. Let us add separate lognormal priors

Lognormal(ln(2), 0.1) and Lognormal(ln(5), 0.1) on each of the two parameters. Note that by adding two priors, we can still treat $\gamma=(\alpha, \beta)$ as single variable having a product Lognormal distribution Lognormal(ln(2), 0.1) x Lognormal(ln(5), 0.1). Thus, our model would be as follows:

Level 1: $X \mid p \sim Bin(100, p)$

Level 2: $p \mid \gamma \sim Beta(\alpha, \beta)$ …$\gamma = (\alpha, \beta)$

Level 3: $\gamma \sim$ Lognormal(ln(2), 0.1) x Lognormal(ln(5), 0.1)

Assuming we have observations for X, using Bayes theorem, the joint posterior distribution can be calculated as,

$$P(\gamma, p \mid X) \propto P(X \mid p, \gamma)\, P(p, \gamma)\, P(\gamma)$$

Using this joint distribution, we can simply marginalize over the non-query variables, to get the posterior distribution of any required variables. Note that although we have used three levels of uncertainty in our example, there is no restriction on the number of levels in a hierarchical model. More layers can always be added in the presence of information about the uncertainty at the top level. Hierarchical modeling thus proposes the additions of as many layers of uncertainty as the information at hand requires to be modeled.

1.2.4 Inverse Reinforcement Learning

Over the last few decades, Reinforcement learning has become a popular learning algorithm for autonomous agents. The main idea here is that the agent forages through the state space and gains rewards or 'reinforcements' while simultaneously learning an optimal policy in the given environment. For example, a bee receives a reinforcement as a function of nectar obtained and distance travelled while foraging in a garden. The initial foraging may be random or

guided by a given policy, but during the learning period, the agent learns the optimal policy either 'actively', where it simultaneously deploys its learnt policy occasionally while learning, or passively, where it just follows the initial policy throughout the learning phase. However, it should be noted here that the agent receives rewards when learning which means that the environment has a reward function hard-coded into it which the agent cannot access directly but sees a part of in each step. Thus, to help such an autonomous agent learn an optimal policy, we must hand-design a reward function purely based on our beliefs about the environment and an arbitrary scale. Hence, reinforcement learning is prone to error and can only be as good as the initial available reward function. For example, the bee's actual reward function may consist of many more parameters like wind speed, predator information and more such variables whose relative magnitudes are only empirically learnt by the bee and cannot be easily determined.

However, from another learning perspective, humans often learn to accomplish certain tasks by observing an 'expert' instead of exploring the environment on their own. Using this natural learning idea, assuming we have knowledge that there is an 'expert' for a given task and that we have observations of the expert's behavior over time in a variety of situations, we can infer the reward function of the expert and treat it as optimal. The learning of reward functions is preferred over learning of policies because the reward function is considered the most robust, succinct and transferable definition of a task[5]. This kind of learning is termed as Inverse Reinforcement Learning (IRL) and has become popular in recent years. IRL was first proposed as a viable technique in 1998[4] and since then many different approaches and algorithms have been proposed for this type of learning.

One of the early papers on IRL[5] proposes three types of algorithms under different conditions of the environment and availability of information. The first algorithm proposes a linear

programming approach for finite state MDPs by characterizing the set of all reward functions under which the expert's policy is optimal. However, this set is provably degenerate as many reward functions exist under which the expert's policy is optimal. For example, a reward function which has the same reward everywhere, i.e. a constant reward function, can be trivially a solution for any given expert policy. This is because the reward is same no matter what action is chosen in any state, and hence all policies are equally optimal including the expert's policy. This problem is solved by choosing the reward function that maximally differentiates between the optimal and other suboptimal policies. Of course, this algorithm and the second algorithm both assume that the expert's entire policy is available, not just expert actions in some of the states. The third algorithm is however much more flexible and does not need the expert policy but just the expert's trajectories. We consider the third approach more useful since usually, we can only observe a realization of the expert's policy in the form of a trajectory rather than the expert's policy itself. Another efficient method to alleviate the degeneracy issue was proposed by Ziebart et al.[6] which proposes a maximum entropy approach wherein among those policies that satisfy the constraints of matching the expert's policy, the one with maximum entropy is the best policy and the reward corresponding to it is the best solution.

Ramachandran et al.[7] proposed a Bayesian approach to IRL in which they describe two types of learning – reward learning and apprenticeship learning. While apprenticeship learning assumes forehand knowledge of the expert policy, reward learning performs estimation of the reward function using a sampling technique called PolicyWalk. The general idea in Bayesian IRL is to compute the posterior of the reward function given the observations of the expert trajectories by choosing appropriate priors over the reward function and performing sampling.

Another interesting paper using a Bayesian approach to perform IRL was proposed by Rothkopf and Dimitrakakis[8] which claims to be able to match or even surpass the performance of the expert in the given task. The idea of surpassing the expert's performance here comes from the assumption that the agent may be acting nearly optimally with respect to its preferences. The authors use a reward-policy model separating the rewards from the policy in that the reward function no more corresponds to a single policy through the value function as it usually does in MDPs. Rather, the reward function defines a distribution over parameterized policies such that given the parameter and the reward function, the policy is uniquely determined. Estimation of the reward functions and policy is done using a Metropolis-Hastings sampler. However, this work was extended to a multitask IRL algorithm by Dimitrakakis et al. which we shall closely follow. Here, the authors propose to determine the various preferences of an agent performing a series of tasks in an environment. This is also equivalent to the inferring the preferences of multiple agents performing the same tasks but with different motivations resulting in different behaviors. Of course, this problem can be solved as a series of multiple IRL problems, one corresponding to each task or expert. But the idea is to connect the different tasks or different expert motivations via structured priors to capture common biases while simultaneously differentiating between their individual motivations.

The algorithm described in this thesis can be considered a slight deviation from the algorithm of Dimitrakakis et al.[9]. In chapter 3, we shall shed more light on this work while we describe our model since it is similar to the model described in [9].

1.3 Contributions

We shall attempt to make the following contributions in this thesis:

- We shall use the NGSIM dataset available to download from the U.S. Department of Transportation website https://ops.fhwa.dot.gov/trafficanalysistools/ngsim.htm to establish that a driver does not drive with one single motivation or reward function when passing through the zone of a freeway near an acceleration ramp. We shall refer to this zone as the 'merging zone'.
- We assume that there exists an 'average behavior' for all drivers passing through the merging zone and hypothesize that this 'average behavior' is motivated by multiple reward functions depending on the section of the merging zone the driver is in.
- Our first main contribution would be to determine the number of sections the merging zone can be split into such that the average behavior in each section is driven by the same reward function while the reward function for each section is different from that of its adjacent ones.
- Our second main contribution would be to determine the individual reward functions and policies of the average driver in each section.

1.4 Related Work

Apart from the Bayesian multitask IRL paper[9], which this thesis closely follows, there has been other related work in the field of Bayesian multitask IRL using different approaches. Babes et al.[10] proposed that when we observe many trajectories from as many experts, we see groups of same behaviors among them which leads to the hypothesis that the actual number of distinct experts is much lower than the large number of expert trajectories observed. In this method, the

many trajectories are first clustered using an Expectation-Maximization approach and then the reward functions for each cluster are computed. Thus, this method requires the number of clusters to be specified as a parameter before the learning process can start. This work was improved upon by Choi et al.[11] who propose a non-parametric approach by the use of a Dirichlet process mixture model and generalize this kind of clustering approach. Another notable approach under the multitask learning category is that of Wilson et al.[12]. This paper, however, deals with Reinforcement Learning(RL) rather than IRL but we mention this in this context since it uses a hierarchical Bayesian model to determine the distribution over possible MDPs assuming that an agent can be put into an arbitrary environment and must quickly learn which environment it is in using the posterior of the distribution over MDPs given various environment parameters.

1.5 Thesis Outline

- Chapter 2 describes the NGSIM dataset and details its metadata. This chapter also later describes how we extract trajectory information relevant to this thesis and generate our MDP model.
- Chapter 3 describes the task separation process and results of the individual trajectories being separated into multiple task trajectories using a split-merge technique similar to a segmentation technique.
- Chapter 4 describes our main Likelihood Weighting IRL algorithm and the results for our multitask IRL.

CHAPTER 2

DATA AND SETUP

2.1 The NGSIM Program

The Next Generation Simulation (NGSIM) program was initiated by the United States Department of Transportation (US DOT) Federal Highway Administration (FHWA) in the early 2000s. The program developed a core of open behavioral algorithms to support traffic simulation with a primary focus on microscopic modeling. It also included supporting documentation and validation data sets that describe the interactions of multimodal travelers, vehicles and highway systems, and interactions presented to them from traffic control devices, delineation, congestion, and other features of the environment. NGSIM stakeholders considered the collection of real-world vehicle trajectory data as important to understand and research driver behavior.

The Interstate 80 (I-80) freeway dataset[13] was one of many datasets collected under the NGSIM program. Researchers for the NGSIM program collected detailed vehicle trajectory data on eastbound I–80 in the San Francisco Bay area in Emeryville, CA, on April 13, 2005. The study was conducted over an area of length 1650 feet or 503 meters and covered six freeway lanes including a HOV lane. The study area included an acceleration ramp. Seven synchronized digital video cameras, mounted on top of a 30-story building adjacent to the freeway, were used to capture vehicle trajectory data. A customized software application called NG-VIDEO developed for the NGSIM program transcribed the video data into text. 45 minutes of data was recorded in three intervals: 4.00 p.m. - 4.15 p.m., 5.00 p.m. - 5.15 p.m. and 5.15 p.m. - 5.30 p.m. These periods

represent the buildup of congestion in traffic in this area. Figure 2.1 shows the span of the freeway covered by each camera and the lanes of the freeway including the on-ramp.

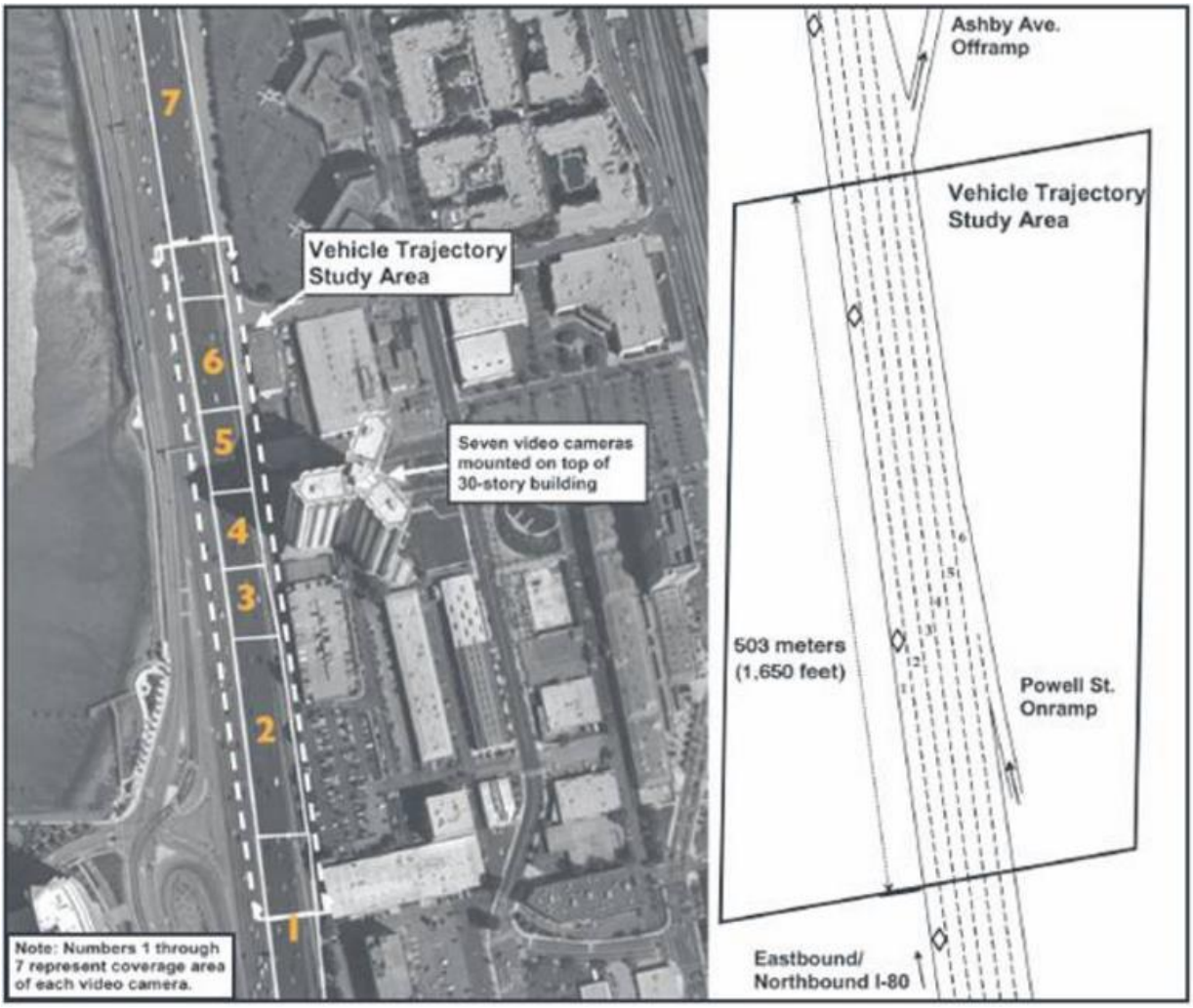

Figure 2.1 Left: Camera-wise coverage area; Right: Lane-diagram showing all freeway lanes including the on-ramp which is not clearly visible in the picture to the left

We shall now see some snapshots of video data that detail the boundaries of each camera's coverage area.

Figure 2.2 Camera 1 coverage

Based on timestamps in the video and the positions of certain vehicles at those times, we infer that camera 1 spans from y=0 feet to approximately y=249 feet. The y coordinate here is measured along the length of the road from left to right. However, vehicle trajectories are only tracked from approximately y=64 feet. We can clearly say from Figure 2.2 that on-ramp vehicles cannot merge within this camera's view.

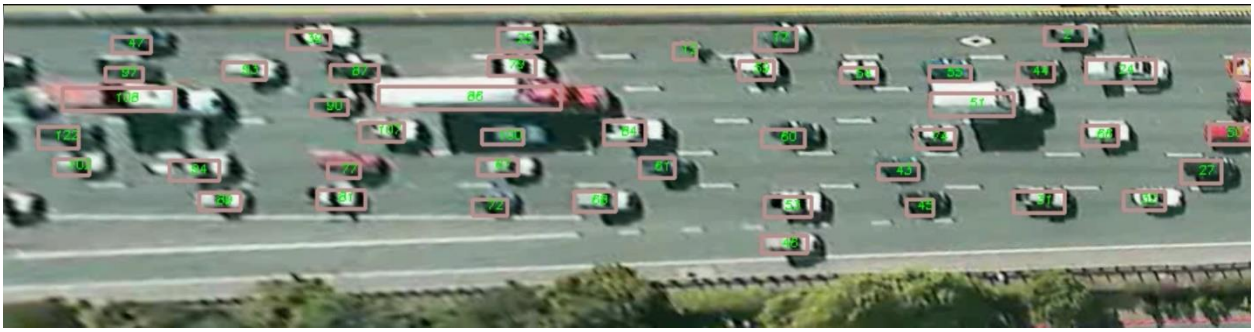

Figure 2.3 Camera 2 coverage

The extent of coverage of camera 2 is approximately y=249 feet to y=714 feet. In Fig 2.3, we see that a little before half the entire span (approximately y = 249+(714-249)/2 = 481 feet) of this camera, vehicles can start legally merging onto the freeway.

Figure 2.4 Left: Camera 3 coverage; Right: Camera 4 coverage

We can see in Figure 2.4 that the merging area ends approximately near end of camera 3's coverage area which is about y=900 feet.

We shall use this information about observed merging section boundaries to compare against the results of our split-merge technique in Chapter 3.

2.2 Metadata description

The transcribed NGSIM dataset is a collection of 3 CSV files, one for each of the three time intervals. Each row consists of 18 columns corresponding to information about one vehicle in one frame. Frame interval is 100ms which means there are 10 rows for every second per vehicle. Following are the most important columns that are of concern to us:

- Column 1: Vehicle ID, unique ID of current vehicle
- Column 2: Frame ID, frames are number from 1, incrementing by one every 100ms. Note that frames do not start at 1 for each vehicle, rather it is a representation of discrete time
- Column 6: Local Y, the longitudinal coordinate of the front center of the vehicle with respect to the entry edge i.e. the start of Camera 1's coverage area measured in feet
- Column 9: Vehicle length, measured in feet

- Column 12: Vehicle velocity, instantaneous velocity measured in feet/second
- Column 13: Vehicle acceleration, instantaneous acceleration measured in feet/second^2
- Column 14: Lane identification, current lane position of the vehicle. Lane 1 is the leftmost HOV lane while lane 7 is the on-ramp
- Column 15: Leading Vehicle ID, ID of the vehicle in front of the current vehicle at that timestamp
- Column 16: Following Vehicle ID, ID of the vehicle behind current vehicle at that timestamp
- Column 17: Spacing, the distance between the front center of the current vehicle to the front center of the preceding vehicle

2.3 Trajectory data extraction

In this section, we shall first describe how we arrive at our MDP model by analyzing the data and then show how we extract trajectory information based on our MDP model.

2.3.1 MDP model

Car A is our agent whose preferences we need to model using IRL. For this first we need to create the MDP model for car A. We choose the following two features to define the state space of car A:

- Spacing (x_{AC}): This is different from the Spacing attribute of column 17 in the dataset. Spacing here is the distance between the front bumper of car A and the rear bumper of the car in front of it i.e. car C. We use this measure because this measure makes more sense if car C is a long vehicle instead. The length of car C, which is the leading vehicle at a given

instant of time, is subtracted from the spacing attribute of column 17 to obtain this feature's value.

- Relative velocity (v_{AC}): We first need to augment the dataset with another column which is the instantaneous velocity of the car C at a given instant of time. Then the relative velocity can simply be computed as the difference between the instantaneous velocity of car A (Column 12) and the instantaneous velocity of car C in the augmented column.

By doing a basic analysis of these features' possible values, we choose the spacing and relative velocity intervals as in Table 2.1 so as to discretize the state space. Also, based on possible instantaneous acceleration values, we choose the acceleration(a) intervals given in Table 2.1 which we shall use to discretize the action space.

x_{AC} (feet)	v_{AC} (feet/second)	a (feet/s^2)
< 0	> 45	< -9
0 – 14	35 – 45	-9 – -4.8
14 – 28	25 – 35	-4.8 – -0.6
28 – 42	15 – 25	-0.6 – 0.6
42 – 56	5 – 15	0.6 – 4.8
56 – 70	-5 – 5	4.8 – 9
70 – 84	-15 – -5	> 9
84 – 98	-25 – -15	
98 – 112	-35 – -25	
112 – 126	-45 – -35	
126 – 140	< -45	
140 – 168		
168 – 196		
> 196		

Table 2.1 Spacing, relative velocity and acceleration intervals

Spacing intervals are in multiples of a standard car length (14 feet as per the data). The first spacing interval of <0 feet may seem absurd as that would mean that cars are on top of one another. But this is only because of outliers in the data due to video transcribing errors. Also, open and closed interval bounds are appropriately used to prevent overlap of intervals. With 13 spacing

intervals and 11 relative velocity intervals, we have a state space of 13x11 = 154 states and an action space of 7 actions. The state space layout is shown in Table 2.2. We shall simply refer to states and actions using integral numbering. However, the actions may also be referred to as High, Medium and Low Braking (Actions 0,1,2), Zero acceleration (Action 3) and Low, Medium and High Acceleration (Actions 4,5,6).

v_{AC} / x_{AC}	> 45	35 – 45	25 – 35	15 – 25	5 – 15	-5 – 5	-15 – -5	-25 – -15	-35 – -25	-45 – -35	< -45
< 0	0	1	2	3	4	5	6	7	8	9	10
0 – 14	11	12	13	14	15	16	17	18	19	20	21
14 – 28	22	23	24	25	26	27	28	29	30	31	32
28 – 42	33	34	35	36	37	38	39	40	41	42	43
42 – 56	44	45	46	47	48	49	50	51	52	53	54
56 – 70	55	56	57	58	59	60	61	62	63	64	65
70 – 84	66	67	68	69	70	71	72	73	74	75	76
84 – 98	77	78	79	80	81	82	83	84	85	86	87
98 – 112	88	89	90	91	92	93	94	95	96	97	98
112 – 126	99	100	101	102	103	104	105	106	107	108	109
126 – 140	110	111	112	113	114	115	116	117	118	119	120
140 – 168	121	122	123	124	125	126	127	128	129	130	131
168 – 196	132	133	134	135	136	137	138	139	140	141	142
> 196	143	144	145	146	147	148	149	150	151	152	153

Table 2.2 State space layout, entries represent state# for a given spacing and relative velocity interval

2.3.2 Trajectory extraction

Now that our state and actions spaces are defined, we shall move on to extracting the trajectories. The data consists of trajectory information for more than 2000 vehicles for each of the three time periods. But since we are only concerned with vehicles in lane 6, we filter and separate frame by frame trajectory information for all vehicles in lane 6. For our purposes, we shall not distinguish between vehicle trajectories based on which time interval they were recorded and ignore congestion differences across the time periods. Thus, we end up with a collection of

trajectories for around 1500 vehicles. We convert these continuous time trajectories to discrete time trajectories by doing a mapping of the spacing, relative velocity and acceleration in each frame. Spacing and relative velocity are mapped to their respective intervals and the intervals together, map to the appropriate state. Similarly, acceleration is mapped to actions and after disregarding irrelevant information, our trajectory files are reduced to three columns; y-position(Column 6 in dataset), state(integer) and action(integer). The y-position is used for task demarcation which will be explained in Chapter 3.

It is noteworthy to mention some of the several reasons for trajectory data loss. For example, sometimes car A demonstrates the absurd behavior of moving to the on-ramp and then merging back into lane 6. This causes a lapse/break in its trajectory as those frames put that car on lane 7 and the data extraction process eliminates that part of the trajectory. This kind of trajectory data loss also happens when car A weaves onto lane 5 and back to lane 6. We shall totally ignore all those parts of the trajectories that do not pass the lane 6 filter.

Finally, we also discuss the transition function generation using the trajectories. We use a simple sampling algorithm based on sampling possible next states given all possible actions for each state. The next state samples are based on motion model calculations in probabilistic robotics[15]. The algorithm is shown in Figure 2.5 and is self-explanatory.

```
function GENERATE-TRANSITION-PROBABILITIES(trajectory_set[])
  returns an array of state transition probabilities
  inputs: trajectory_set, an array of trajectories
  local variables: s′, next sample state
                   a_A, sampled acceleration value based on an acceleration interval
                   vf[s,a,s′], visitation frequency of s,a,s′
  global variables: actions[], set of all possible actions as an ordered list 0,1…,(|A|-1)
                    acc_interval_ends[], set of end points of discretized acceleration, in
                    our case this array is [-11.2, -9, -4.8, -0.6, 0.6, 4.8, 9, 11.2]
                    t, time interval of 0.1
  for each trajectory t in trajectory_set
      s ← trajectory_set[t].state
      for each action a in actions
          for 1000 iterations
              a_A ← random(acc_interval_ends[a], acc_interval_ends[a+1])
              s′[V_AC] ← s[V_AC] + a_A*t                  // Sampled Relative velocity
              s′[X_AC] ← s[X_AC] - s[V_AC]*t − (1/2) a_A*t²  // Sampled Spacing
              vf[s,a,s′] ← vf[s,a,s′] +1
          endfor
      endfor
      for each action a in actions
          for each s′ in all states
              tp[s,a,s′] ← vf[s,a,s′] /sum(vf[s,a,:])
          endfor
      endfor
  endfor
  return tp
```

Figure 2.5 Generating transition probabilities by sampling

CHAPTER 3

TASK SEPARATION

This chapter focuses on our first main contribution where we seek to establish that a driver passing through a merging zone, i.e. a length of a freeway near a ramp, performs not just one but a series of tasks. Our main aim here is to determine the number of tasks (m) and find the boundary positions of each task along the length of the road.

3.1 Split-Merge Clustering

Split-and-Merge segmentation is a well-known technique in the context of computer vision wherein we separate sections of a given image based on optimal homogeneity within a section and heterogeneity between adjacent sections. The optimality criteria based on expert judgement is often hand-tuned by a human.

The basic idea is to iteratively perform a split step based on some splitting criteria followed by repeatedly merging adjacent split sections based on some merging criteria. The whole split and merge is performed repeatedly until convergence which happens when the clusters do not change in successive iterations. The split step segments the ‘whole’ into small pieces whenever a given piece is not considered homogenous enough based on the split criteria. This step essentially over segments with a stronger criterion while the merge steps that follow use a weaker criterion to allow merging. We shall now explain this in more detail for our specific domain instead of a generic whole.

We start by reducing the area of interest on the freeway to 0-1200 feet based on observation. The lower limit is chosen based on where the earliest trajectories start from. The upper limit is simply a nice round figure chosen to be far enough from the merging zone which includes enough extra road-length after the on-ramp ends to cover any possible post-merge behaviors. Also, the upper limit was chosen since it seemed that beyond it, the driving would simply be generic freeway driving which is not of much interest to us.

The next step is to divide the whole merging zone into very small atomic sections like pixels in an image. We choose a length of 2.5 feet as the length of the atomic sections. The set of all these atomic sections makes up the initial sections cluster shown in Figure 3.1. The single cluster consisting of many atomic sections which is analogous to a single undivided image consisting of pixels. Each section has a set of action distributions, one for each state. The set of action distributions can be thought of a measure of the section like intensity of a pixel. We will shortly explain comparing similarity between sections.

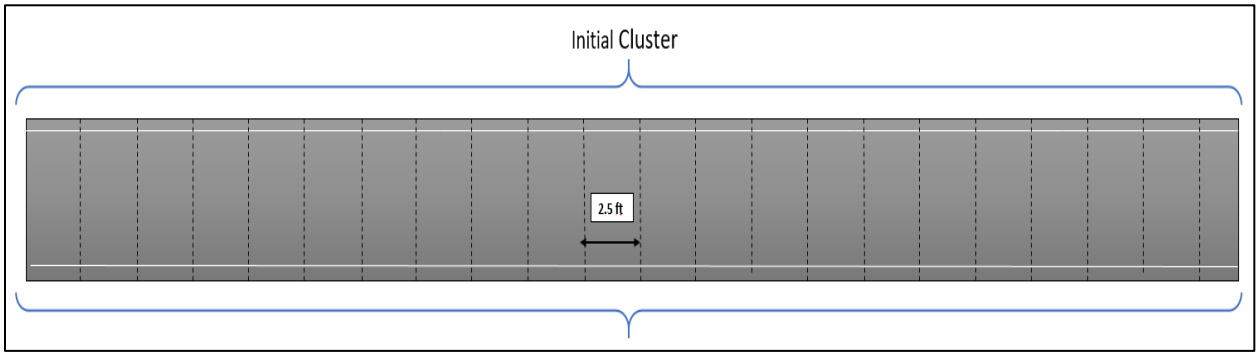

Figure 3.1 Initial cluster for split-merge clustering

We begin each outer split-merge iteration by checking homogeneity of each cluster. To do this we compute the cluster mean which is simply the action distribution obtained by merging all

the sections in the cluster. We now need to measure the average distance of each section from the cluster mean for which we need a distance measure comparing the action distribution sets of the mean and the section. A good symmetric distance measure between probability distributions is the Hellinger distance and for two discrete probability distributions $P=(p_1,p_2\ldots,p_k)$ and $Q=(q_1,q_2\ldots,q_k)$, it is given as

$$H(P,Q) = \frac{1}{\sqrt{2}}\sqrt{\sum_{i=1}^{k}\left(\sqrt{p_i}-\sqrt{q_i}\right)^2} \qquad \ldots \text{Eq } 3.1$$

Since we have one distribution for each state, we shall take an average of Hellinger distances between corresponding states of both sections. Finally, the average distance of each section from the cluster mean is the average of the Hellinger distances of all sections from the mean. We use a empirically decided threshold, ϵ, and split any cluster into two equal halves if the average Hellinger distance exceeds epsilon. This process is repeated recursively for each half whenever a split is performed. When we can no more split any cluster, we end up with a number of clusters as shown in Figure 3.2 who are assumed to be homogenous. The choice of epsilon is selected so that splits are performed aggressively, that is, we need to ensure that we have over-split the cluster more than required. This prepares the clusters for the next merge step. Notice here that each cluster always has contiguous sections and they together form a larger, continuous section of the freeway.

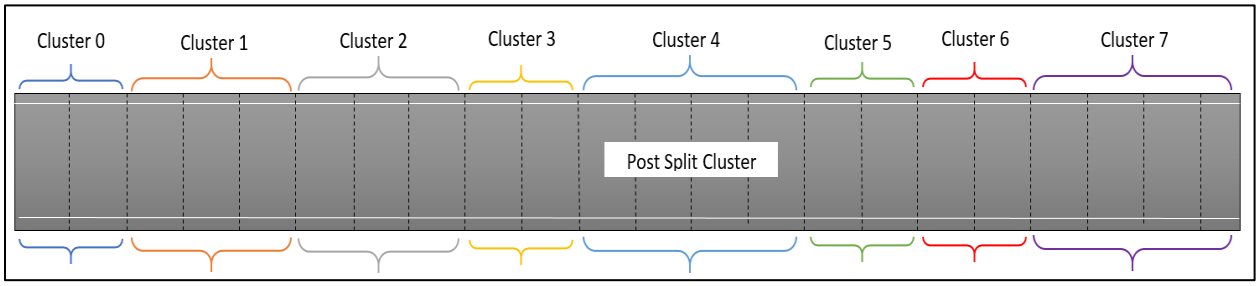

Figure 3.2 Example of post-split clusters

Merging is performed iteratively among consecutive clusters. Consecutive clusters are those, which when merged form a larger continuous cluster representing an unbroken span of the freeway. The merging is done using the same measure as splitting, however, with a more relaxed threshold, δ. We have used $\epsilon = 0.2$ and $\delta=0.32$. In each iteration of merging, for each cluster i, a decision is made whether to merge the i^{th} cluster with the $(i+1)^{th}$ cluster or not. The process stops when no merges are performed in an iteration after which we go to the next outer split-merge iteration. Post-merge clusters formed, after one split-merge iteration, by merging some of the clusters in Figure 3.2 are shown in Figure 3.3. Thus, the split process is recursive, merge process is iterative and the whole split merge process continues iteratively till no cluster changes happen in the outer split-merge loop.

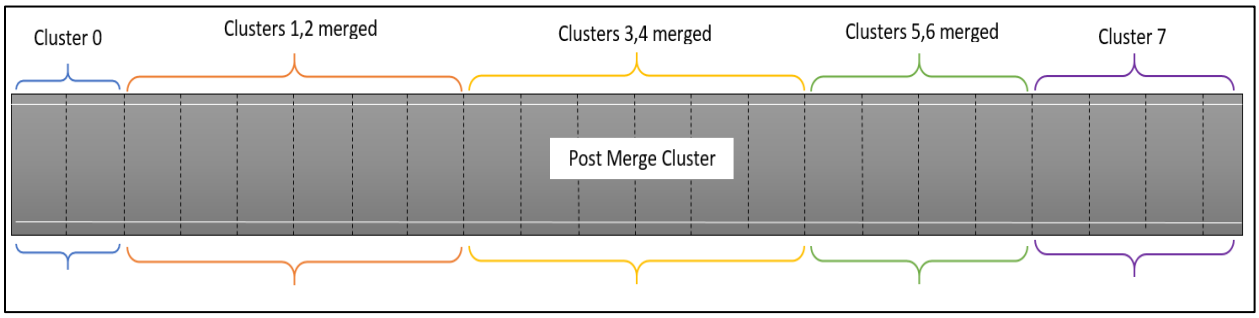

Figure 3.3 Example of post-merge clusters

3.2 Results

This process is expected to be affected by noisy data and when it comes to average of probability distributions' distances it is a measure which could lead to results which would not be as concise as expected. Each final cluster spans from the start-point of its first section and ends at the end-point of its last section. Henceforth, in this chapter and the next, each cluster representing a section of the freeway will be referred to as a 'task section' or simply a 'section'; the number of

task sections thus corresponding to the number of tasks. Table 3.1 shows the cluster boundaries of the final clusters.

Task section#	Bounds(y-position on freeway in feet)
0	0 – 67.5
1	67.5 – 437.5
2	437.5 – 440
3	440 – 830
4	830 – 840
5	840 – 1182.5
6	1182.5 – 1190
7	1190 – 1200

Table 3.1 Resulting task sections from split-merge clustering

Now, based on the following observations, we adjust the cluster boundaries to those shown in Table 3.2 for the sake of convenience in the next section:

- Section 0 (0-67.5 feet) is very sparse, i.e. very few trajectories have state-action pairs in section 0. This is because video transcribing for most vehicles was done starting somewhere in the middle of camera 1's entire coverage area, i.e. somewhere around the 60 feet marker. Thus, we merge it with the next section.
- Section 2 is a small 2.5 feet section. Hence, we conveniently merge it with the previous section.
- Section 4 is again a relatively small 10 feet long section which we merge into the previous section.
- Sections 6 and 7 are also individually quite small and this is the ending part of the freeway under our consideration, so treating it as a harmless outlier, we merge it with Section 5.

Task section#	Bounds(y-position on freeway in feet)
0	0 – 440
1	440 – 840
2	840 – 1200

Table 3.2 Final task sections and their boundaries

It is important to note that such small adjustments should hardly affect the results of the next step because, at worst, these would simply be considered as outliers during Bayesian inferencing and should not affect the end results largely. Thus, we end up with three distinct tasks which we shall conveniently refer to as ‘pre-merge’, ‘merge’ and ‘post-merge’ tasks.

3.3 Correctness of Results

We shall compare the results to some basic measures to show that the results are in accordance to what can be expected based on the data.

The average velocity and average instantaneous acceleration of all cars at any position of the road show trends seen in Figure 3.4.

- Average acceleration has a couple of high spikes at just over 400 feet
- At around 500 feet, the average velocity starts showing an increasing trend instead of the falling trend show until then.
- Average acceleration shows minute variations until around 800 feet after which it starts fluctuating again. However, velocity shows no major change in trend around 800 feet.

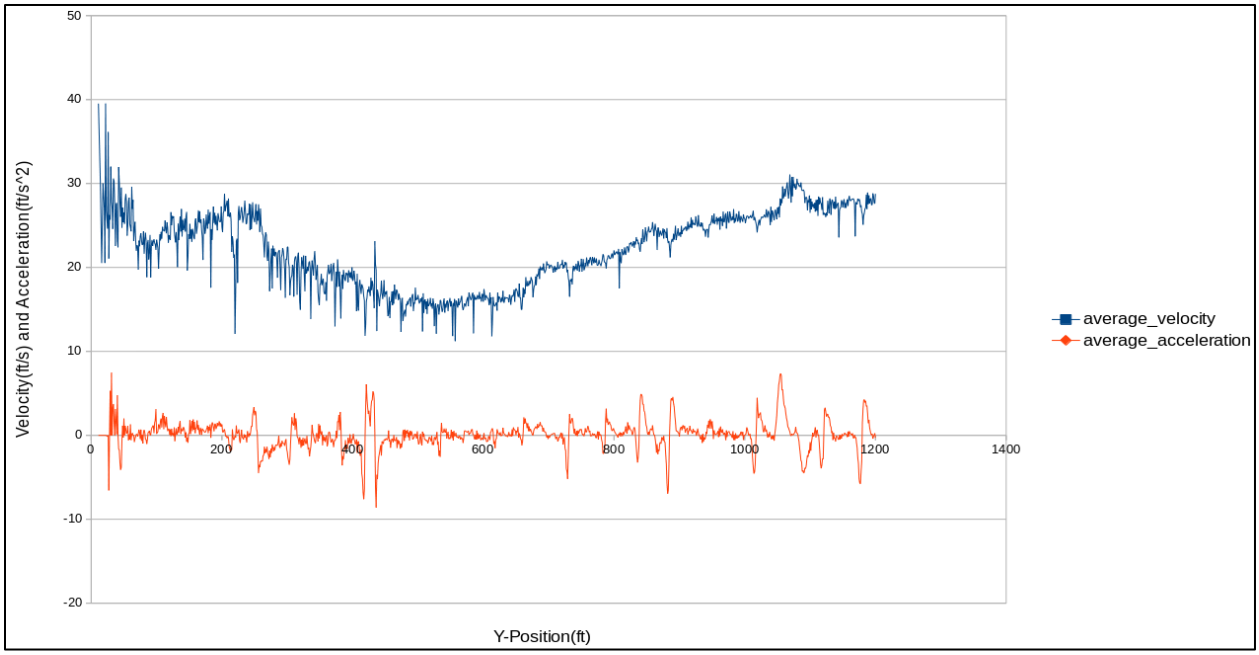

Figure 3.4 Average velocity and average instantaneous acceleration along the road

Thus, we can say that our conclusions of task separation are quite good. It is likely that these results were not perfect because of unpredictability and noise in the recorded behaviors and thus, our estimate is a decent one which we can use for our next stage.

CHAPTER 4

LIKELIHOOD WEIGHTING MULTITASK IRL

Likelihood weighting, as discussed in Chapter 1 is a simple Monte Carlo process used to determine the posterior probability of any unknown random variable(s) in a Bayesian network given evidence data. For our purpose, we shall define a multitask reward-policy model similar to that of [9] to infer the set of three reward functions and the policies of car A given trajectories for car A for each of the three tasks inferred in Chapter 3. This chapter details the multitask IRL algorithm, its minute details and its application to the dataset. The results are summarized in the last section.

4.1 Hierarchical Bayesian Multitask Model

4.1.1 Basics and Notations

We assume that all the dynamics of the environment of car A are constant, no matter what task is being performed. We define these dynamics with the help of a Controlled Markov Process $v = (S, A, T, \gamma)$. The state space, S, and action space, A, were defined in Chapter 2 along with the method to generate the transition function T. γ is usually chosen as a value close to 1 hence we set $\gamma = 0.99$. Since, the environment is Markovian, if car A is in state s_t at time t and performs action a_t, then the next state s_{t+1} has distribution $T(S| s_t , a_t)$. We shall denote the trajectory generated by the car over time t as $(s^t, a^t) \equiv \{(s_1, a_1), (s_2, a_2)\ldots (s_t, a_t)\}$ with each action a_t chosen from its optimal policy $\pi(A | s_t)$. Since we have m = 3 tasks performed by the car, we have three sets of trajectories and since we have N_m trajectories per task, we shall denote the n^{th} trajectory of task m as $d_{m,n} \equiv$

$(s_{m,n}^{T_{m,n}}, a_{m,n}^{T_{m,n}})$ where $T_{m,n}$ denotes the length of that trajectory. For simplicity of notation and unless mentioned otherwise, we shall simply assume that there is only one trajectory per task denoted as $d_m \equiv (s_m^{T_m}, a_m^{T_m})$ and later generalize for N_m trajectories in the algorithm. The trajectories form the evidence for Bayesian inferencing as shown later.

Let the reward function and policy of the car for each task be denoted by ρ_m and π_m respectively. Here, by assuming that π was not computed using just ρ, we assert that the car A's policy may not be optimal [9]. Consequently, there is the chance to find policies better than the expert's policy. For simplicity, we drop the m notation when we are not comparing tasks against one another. Let μ be the MDP induced by using reward ρ in CMP ν. μ and ρ can be used interchangeably since *v* remains constant. Given μ and π, the likelihood of a trajectory $d \equiv (s^t, a^t)$ is given by

$$P((s^t, a^t) \mid \mu, \pi) = P((s_1, a_1), (s_2, a_2) \ldots (s_t, a_t) \mid \mu, \pi)$$

$$= T_\mu(s_1|s_0, a_0) * \pi(a_1|s_1) * T_\mu(s_2|s_1,a_1) * \pi(a_2|s_2) \ldots * T_\mu(s_t|s_{t-1},a_{t-1}) * \pi(a_t|s_t)$$

$$= \prod_{i=1}^{t} T_\mu(s_i|s_{i-1}, a_{i-1})\, \pi(a_i|s_i) \qquad \ldots \text{Eq } 4.1$$

where $T_\mu(s_1|s_0, a_0) = T_\mu(s_1)$ which is the initial state distribution.

Also, we shall introduce the optimal Q-function $Q_\mu^*(s, a)$ for MDP μ. Computation of the Q-function given μ is explained in Chapter 1 and given by Eq. 1.4. As we can see in Eq. 4.1, to compute the likelihood of a trajectory given a reward function and a policy we need to compute a stochastic policy for car A given the MDP μ since a deterministic policy could potentially reduce the likelihood to zero with just one wrong action for a given state. Also, we need to parameterize the stochastic policy given a reward ρ so that we can assume a prior over the parameter later. The

Q-function can be used to compute a parametric stochastic policy given a reward function using the soft-max function with an inverse temperature parameter c.

$$\pi(a_i|s_i, \mu, c) = \text{Softmax}(a_i|s_i, \mu, c)$$

$$= \frac{e^{cQ^*_\mu(s_i,a_i)}}{\sum_{a\epsilon A} e^{cQ^*_\mu(s_i,a)}} \quad \text{... Eq. 4.2}$$

With the means to compute likelihood of the evidence for a given reward function and policy our problem can be defined as the estimation of reward functions ρ_1, ρ_2, ρ_3 and policies π_1, π_2, π_3 given the sets of trajectories D = $d_1^{N_1}, d_2^{N_2}, d_3^{N_3}$. We now move to description of the hierarchical model.

4.1.2 Model description

Let $\mathscr{R}$ be the space of reward functions of car A. This space remains the same for reward functions of the three tasks. Let $\psi(\rho) \equiv P(\mathscr{R} \mid \alpha)$ be the prior over the space of reward functions $\mathscr{R}$ with parameter α. Let $\mathcal{P}$ be the space of policies of car A. Since we reduced the policy space given the reward function ρ to a parametric form parameterized by c, therefore, $\pi \equiv f(c, \rho) \epsilon \mathcal{P}$. Now if $c \epsilon \mathbb{R}^1$, then let $\xi(c) \equiv P(\mathbb{R}^1 \mid \beta)$ is a prior probability measure over the real number space parameterized by β. Now, if $\mathcal{A}$ is the space of α and $\mathcal{B}$ is the space of β, then the cartesian product $\mathcal{A}$ x $\mathcal{B}$ denotes the product space over the reward and c parameter priors. Finally, we define $\eta \equiv (\eta_\alpha, \eta_\beta)$ to be the product hyperprior over $\mathcal{A}$ x $\mathcal{B}$ such that $\eta(\alpha, \beta) = \eta_\alpha(\alpha) \cdot \eta_\beta(\beta)$ where $\eta_\alpha(\alpha) \equiv P(\alpha \mid \eta_\alpha)$ and $\eta_\beta(\beta) \equiv P(\beta \mid \eta_\beta)$. The complete Hierarchical Bayesian model for three tasks is shown in Figure 4.1. All variables except η, which we choose based on our beliefs, and d_1, d_2, d_3, which are evidence variables, are latent variables.

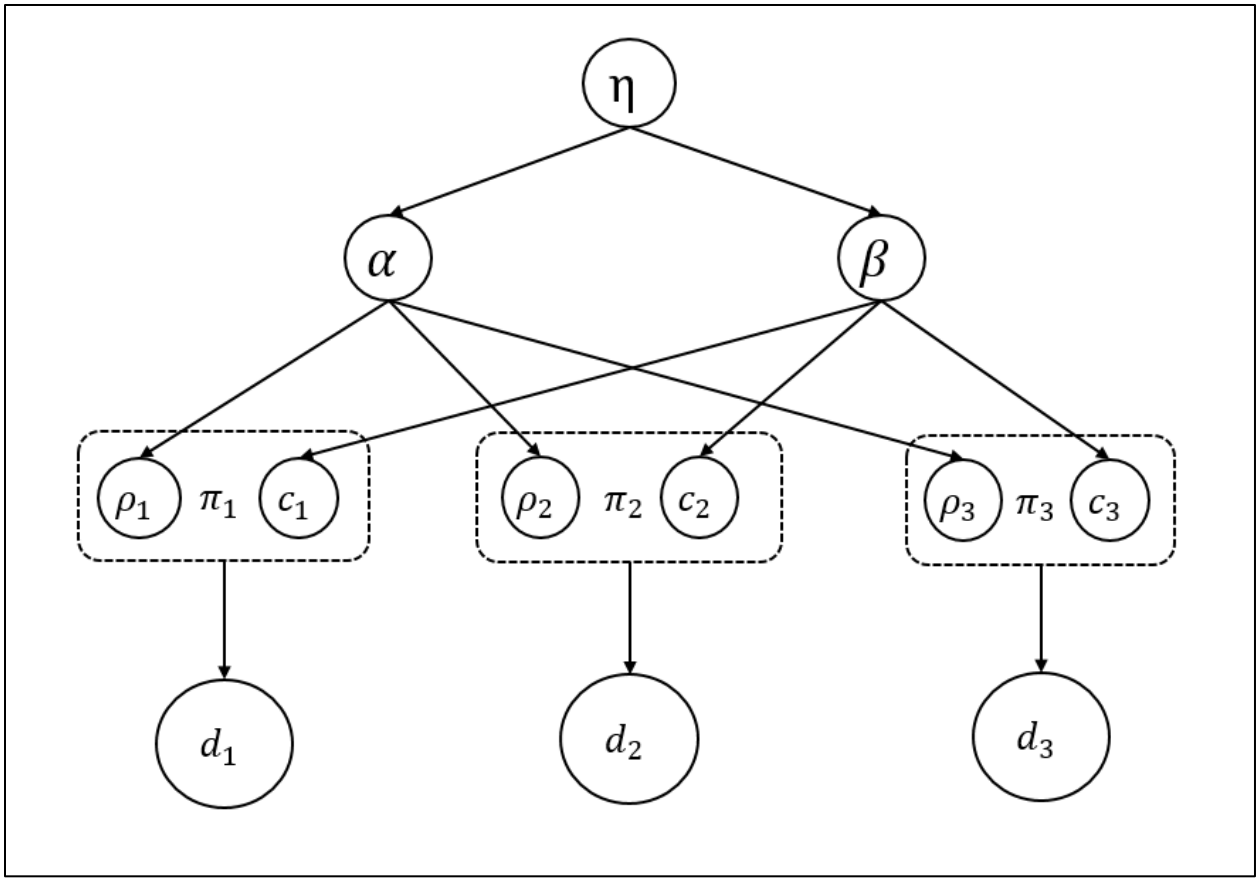

Figure 4.1 Hierarchical Bayesian Multitask model for 3 tasks where D=(d_1, d_2, d_3) is the evidence from the dataset

4.2 Choice of Priors

In this section, we shall explain our choices of priors and the motivation behind those for each of the 4 variables - ρ, c, α, β.

4.2.1 Prior over c : Soft-max Prior

Before we select a prior over c, we shall look at the effect of the c parameter on the policy induced by the optimal Q-function. Consider the sample optimal Q-function values given for some arbitrary state s and for all actions shown in Table 4.1.

a	Q*(s, a)
0	30.3
1	30.2
2	30.4
3	31
4	30.5
5	30.8
6	30

Table 4.1 Sample Q*(s, a) values for some state s

Now, for some values of c ranging from 0.1 to 20, Figure 4.2 gives the stochastic policy generated using Eq. 4.2 and the Q-values in Table 4.1.

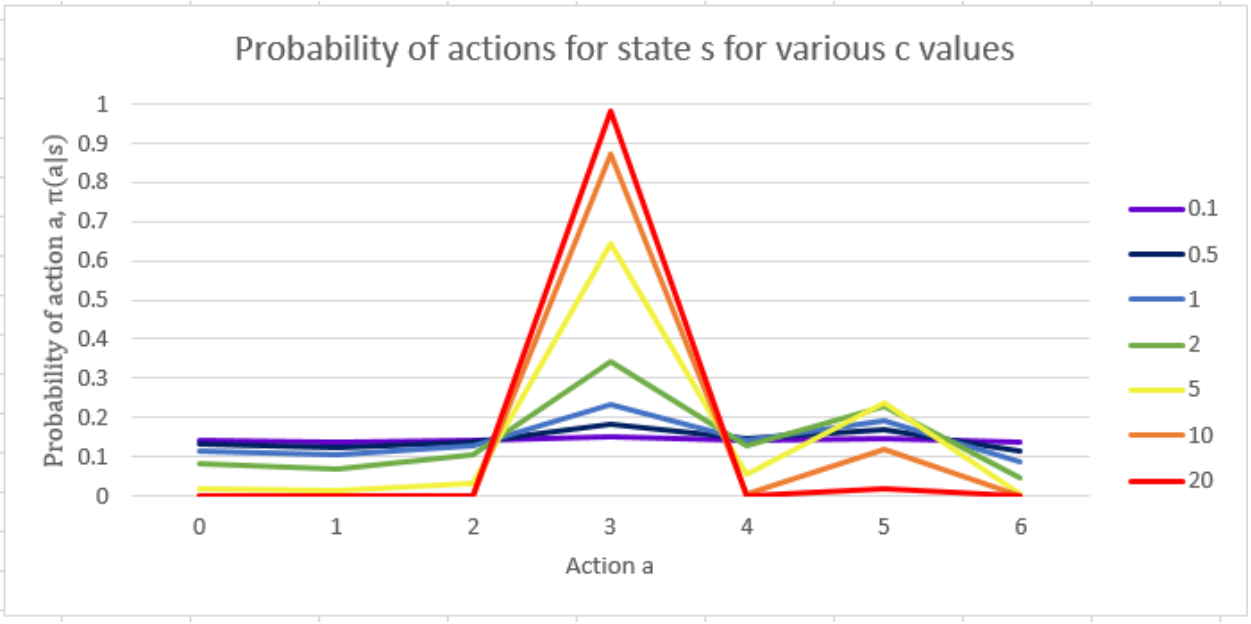

Figure 4.2 Stochastic policy for some state s using various c values

We see in Figure 4.2 that for higher values of c, the difference in action probabilities is amplified thus giving more 'peaky' stochastic policies while for lower values of c, the policy becomes flatter,

i.e., ignorant of the Q-values. Also, too high c values tend to magnify only the action probability corresponding to the highest Q-value while causing the rest of the action probabilities to become equal and lose relative differences. Hence, we shall use an exponential distribution with parameter λ=0.2. The expected value of this distribution = $1/\lambda$ = 5, which gives good policies for Q-values of similar order of magnitude as that of Table 4.1. While it may seem a viable option to choose a log-normal distribution centered around 5 instead(μ=ln(5)), this would restrict the values based on the choice of variance.

4.2.2 Prior over the Exponential c-prior : Soft-max hyperprior

Now, instead of using an Exponential distribution with λ=0.2 as mentioned in Section 4.2.1, we shall use another Exponential distribution with expected value λ^{-1} = 0.2 to model the distribution of the parameter of the Soft-max prior. Thus, the Exponential distribution with λ = 5, with mean value 0.2, shall be used as a hyperprior over the c parameter. Thus, η_β =5.

4.2.3 Prior over ρ : Reward prior

To understand the Product Dirichlet prior over ρ, we shall first build some intuition for the Dirichlet distribution. The Dirichlet distribution is used to represent a distribution over distributions. It takes k parameters $\alpha = (\alpha_1, \alpha_2 \ldots, \alpha_k)$ and generates a distribution over the open (k-1)-dimensional simplex. A sample $x = (x_1, x_2 \ldots, x_k)$ from such a distribution is constrained by $\sum_{i=1}^{k} x_i = 1$ and thus represents a multinomial distribution. Figure 4.3 shows two k=3 Dirichlet distributions on the 2-simplex with different parameters.

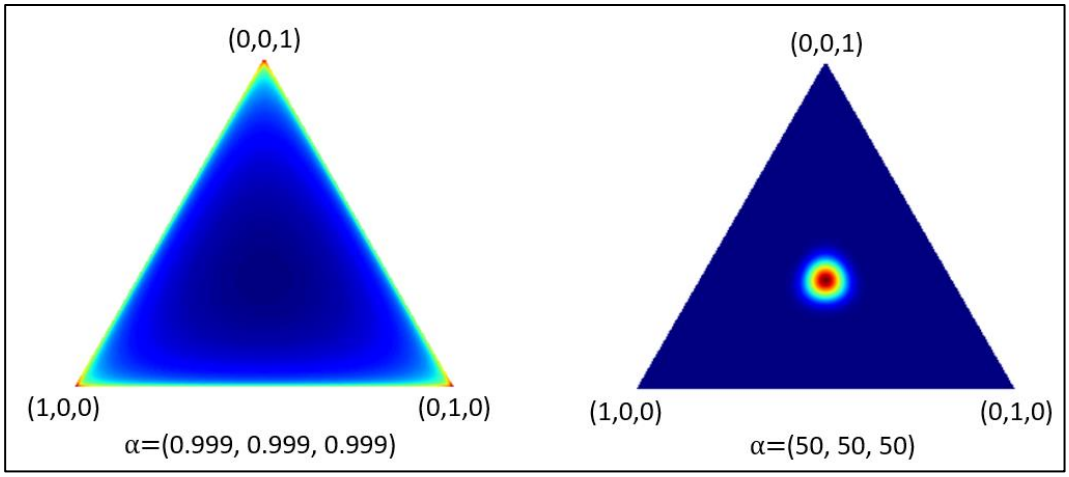

Figure 4.3 Dirichlet distributions with dark blue areas with lowest probabilities and red areas with highest probabilities.

Although it is possible to have different values for each of the k α-parameters, we shall restrict our Dirichlet distributions to those with same α-parameters. This is because in case of unequal parameters, the distributions skews towards some corner of the simplex and is no more symmetric and we claim to not know such biases in the reward function beforehand.

Let |S| and |A| denote the number of states and actions, respectively. Now for the reward prior, we shall first sample |S| different alpha values independently, one for each state. We then duplicate the alpha, |A| times for every state to generate a |A|-dimensional vector of alpha parameters for each state. At this point, the vector of alpha parameters corresponding to each state represents the set of Dirichlet parameters of the distribution over the actions of that state. Then, one sample is drawn from each of the |S| Dirichlet distributions. The matrix of all the samples represents a reward function. Thus, a reward function is such that for each state s, the reward values represent a multinomial distribution over the actions. Since every multinomial distribution corresponding to each of the states is independently sampled from a different Dirichlet distribution, a reward function is a sample from the Product Dirichlet distribution.

4.2.4 Prior over the Product Dirichlet prior : Reward hyperprior

The first step in sampling from the reward prior as mentioned in Section 4.2.4 involves sampling of |S| different alpha values. Now, if the alpha values are large, then we get a distribution like the distribution with α=(50, 50, 50) shown in Figure 4.3 where the central high probability region corresponds to the space of multinomial distributions $x = (x_1, x_2 \ldots x_k)$ such that $x_1, x_2 \ldots, x_k$ are all almost equal i.e. uniform multinomial distributions are preferred mostly. This is not ideal since in real life problems, MDPs have parsimonious reward structures [7] with a few actions dominating in each state. Thus, we need a steeper Exponential distribution to model the prior over the alpha parameters. We shall choose the exponential distribution with λ=10 with mean value 0.1 as the reward hyperprior distribution. Thus, $\eta_\alpha = 10$ ensures that the α values are mostly <1, thereby, ensuring that the sampled reward functions have few dominating actions for each state and the distribution over actions for a given state is mostly not flat.

4.3 Likelihood Weighting IRL Algorithm for Multitask Reward Calculation

The likelihood weighting IRL algorithm is similar to the general likelihood weighting and parallels can be easily drawn between them. The general idea of sampling, computing likelihood weight of a sample are the same. At the end, we compute the expected value of the three reward functions from the normalized likelihood weights. Such algorithms are known to converge asymptotically to the expected value under certain basic assumptions [14] but computing bounds on the number of samples requires stronger technical assumptions.

4.3.1 Algorithm

Figure 4.4 shows the main algorithm which involves calls to other procedures which have been detailed in subsequent Figures 4.5 and Figure 4.6 for completeness.

LWBMIRL(D, hbn, K)

returns E[$\rho_1, \rho_2 \ldots, \rho_m$], the expected set of m reward functions

E[$\pi_1, \pi_2 \ldots, \pi_m$], the expected set of policies for each of the m tasks

inputs: D = ($d_1, d_2 \ldots, d_m$), the collection of trajectory sets for each of the m tasks

hbn, the hierarchical Bayesian network consisting of η_α, η_β and the CMP v

K, the number of samples to be taken

local variables: RC, vector of reward combinations seen across all samples

C, vector of c parameter combinations corresponding to the reward combinations in RC

W, vector of weights of reward combinations in RC

$\widehat{\rho_1}, \widehat{\rho_2} \ldots, \widehat{\rho_m} \leftarrow$ Intialize reward vectors of m tasks to zero vectors

$\widehat{c_1}, \widehat{c_2} \ldots, \widehat{c_m} \leftarrow$ Intialize reward vectors of m tasks to zero vectors

for k=1 to K

(s,w) ← WeightedMCSample(hbn, D)

RC[k] ← s.rewardset

C[k] ← s.cset

W[k] ← e^w

endfor

Normalize(W)

for k=1 to K

for i = 1 to m

$\widehat{\rho_i}$ += RC[k][i] * W[k]

$\widehat{c_i}$ += C[k][i] * W[k]

endfor

endfor

for i = 1 to m

$\mu_i \leftarrow$ (hbn.v, $\widehat{\rho_i}$)

$Q_i^* \leftarrow$ ComputeQUsingVI(μ_i)

$\widehat{\pi_i} \leftarrow$ SoftmaxPolicy(Q_i^*,$\widehat{c_i}$)

endfor

return (($\widehat{\rho_1}, \widehat{\rho_2} \ldots, \widehat{\rho_m}$), ($\widehat{\pi_1}, \widehat{\pi_2} \ldots, \widehat{\pi_m}$))

Figure 4.4 Likelihood Weighting Bayesian Multitask IRL algorithm

WeightedMCSample(hbn, D)

returns a Sample s alongwith its log weight w

inputs: hbn, the hierarchical Bayesian network consisting of η_α, η_β and the CMP ν

D = (d_1, d_2…, d_m), the collection of trajectory sets for each of the m tasks,

local variables: $\bar{\alpha}'$, vector of |S| elements each representing the Dirichlet parameter for state S

α, Matrix of all alpha parameters of size |S| × |A| elements

s, The sample consisting of the reward and c parameter sets

$\beta \sim \text{Exp}(\text{hbn}.\eta_\beta)$

$\bar{\alpha}'[s] \sim \text{Exp}(\text{hbn}.\eta_\alpha)$, ∀s

$\alpha[s][a] = \bar{\alpha}'[s]$, ∀s ∀a

logw = 0

for i=1 to m

$c_i \sim \text{Exp}(\beta)$

s.cset[i] ← c_i

$\rho_i[s] \sim \text{Dir}(\alpha[s])\ \forall s$

s.rewardset[i] ← ρ_i

$\mu_i \leftarrow (\text{hbn}.\nu\ ,\ \rho_i)$

$Q_i^* \leftarrow$ ComputeQUsingVI(μ_i)

$\pi_i \leftarrow$ SoftmaxPolicy(Q_i^*,c_i)

lw=0

for each d in d_i

for each (s_t , a_t) in d

lw += log $\pi_i(a_t|\ s_t)$

endfor

endfor

logw += $\frac{lw}{|d_i|}$

endfor

return (s, logw)

Figure 4.5 WeightedMCSample algorithm

ComputeQUsingVI (μ)

returns the optimal Q function for MDP μ

inputs: μ, the mdp

local variables: ϵ, The maximum allowable error parameter used for Value iteration

$\epsilon \leftarrow 0.1$

$V^* \leftarrow$ VALUE-ITERATION(μ , ϵ)

$Q^*(s,a) \leftarrow R(s,a) + \sum_{s'} T(s,a,s') * V^*(s') , \forall s \forall a$

return Q^*

Figure 4.6 ComputeQUsingVI algorithm

The WeightedMCSample algorithm is the core sampling algorithm which draws samples from the joint distribution of the Hierarchical Bayesian network. ComputeQUsingVI is a basic algorithm which uses Value iteration to compute the optimal Q function for a completely specified MDP.

4.3.2 Algorithm Discussion

Appendix A gives an importance sampling perspective of likelihood weighting and shows how likelihood weighting is simply a special case of it. As we see in the WeightedMCSample algorithm, the likelihood weight calculation method does not include the transition function term despite Eq 4.1 saying otherwise. It can be shown that the product of all transitions in all the trajectories of all tasks is a constant multiplier to the likelihood weight of all samples. This constant is eliminated during the normalization of weights and hence need not be computed in the first place.

Practically, the W[k] = e^w step in the LWBMIRL which converts the weights back to the linear scale cannot be performed due to the possibility of underflow. Moreover, the computation

of the sum of such weights is even more prone to underflow. The LogSumExp technique can help circumvent this issue when dealing with very small weights.

The total sum of log weights for all trajectories of a task is divided by the number of trajectories in that task. This averaging of the log weights over trajectories translates to a $(1/N_m)^{th}$ root of the weights over each task in the linear scale where N_m is the number of trajectories in the m^{th} task. The reason for the averaging is because the tasks do not have the same number of trajectories due to which the task(s) having more trajectories may influence the overall sample weight more than the task(s) with fewer trajectories.

4.4 Results

The results of this approach applied to this domain are detailed in this section. We shall first show the expected reward functions obtained in many runs of 500,000 samples and then move to the more conclusive expected policies.

4.4.1 Expected Rewards

Figure 4.8 shows the expected rewards for the three tasks obtained from one run of 500,000 samples. The reward functions are arranged as matrices such that rows represent states and columns represent actions. Thus, each entry (i, j) in a matrix corresponds to R(i,j) i.e., the reward of performing action j in state i. The reward values for a given state are color coded using the scale in Fig 4.7.

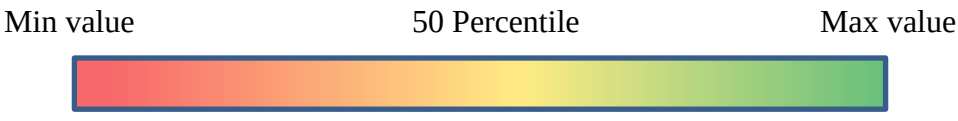

Figure 4.7 Reward Function scale for each state

Looking at the rewards, we can certainly see a lot of dissimilarities in the three reward functions but making a definitive conclusion that the reward functions are correct is a little difficult.

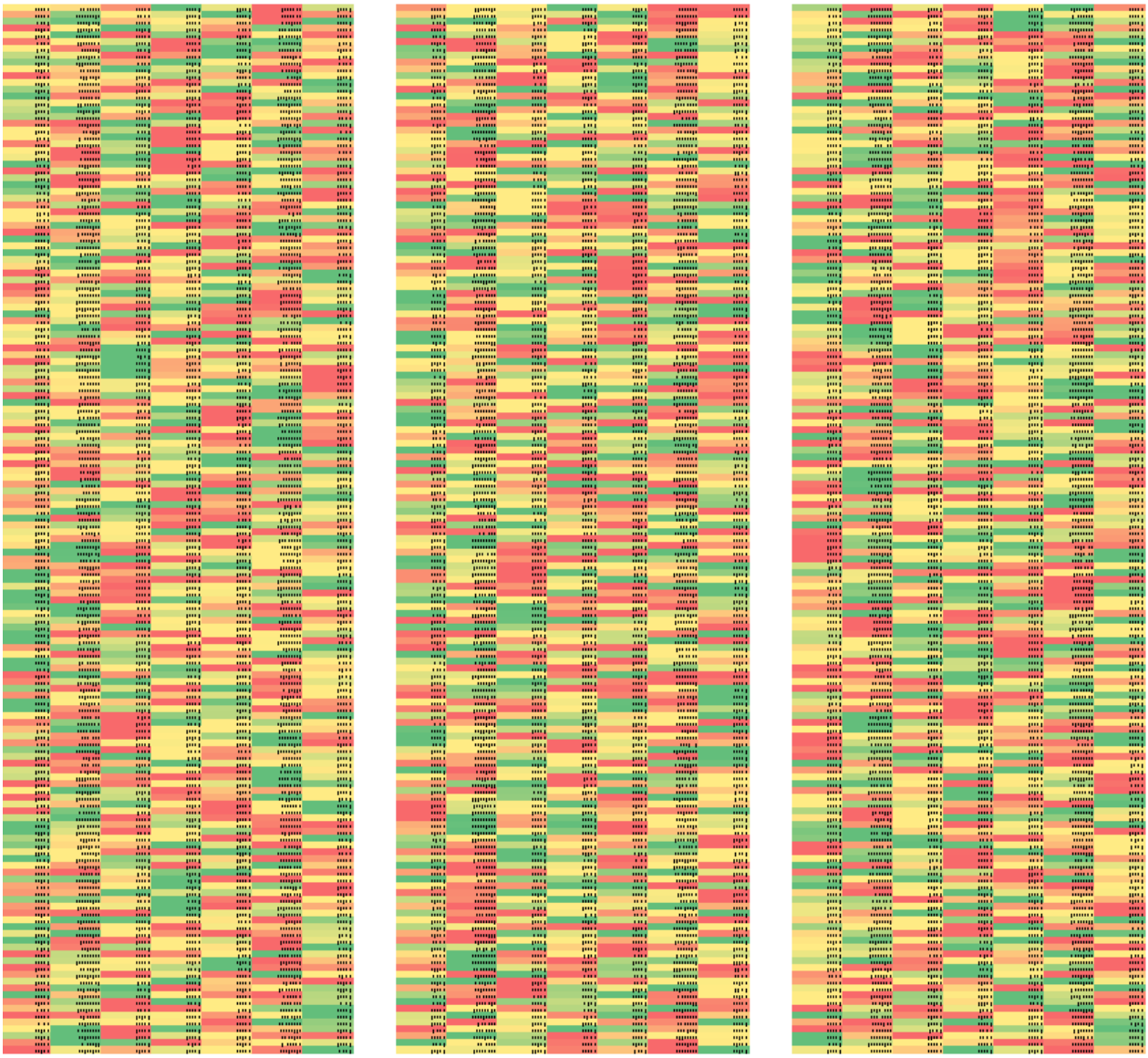

Figure 4.8 Expected Rewards for a single run of 500K samples

4.4.2 Expected Policies

Figure 4.10 shows the expected stochastic policies corresponding to the three reward functions shown in Figure 4.8. The policies are arranged in the same manner as the reward function. Each value (i, j) in a policy matrix corresponds to the probability of taking the action j in state i. The policy values for a given state are color coded using the scale shown in Figure 4.9.

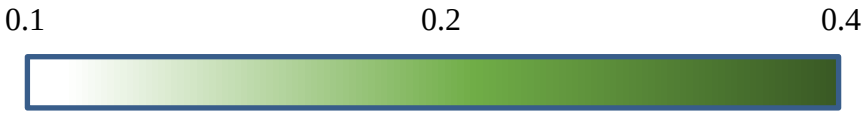

Figure 4.9 Policy scale for each state

In Figure 4.10 we definitively see differences between the behaviors of cars in the 1st and 2nd sections and 2nd and 3rd sections. Based on the above policy, in the merging section, drivers of car A would be more likely to brake when the spacing between car A and its leading car is less and more likely to accelerate when there is opportunity to do so with more spacing.

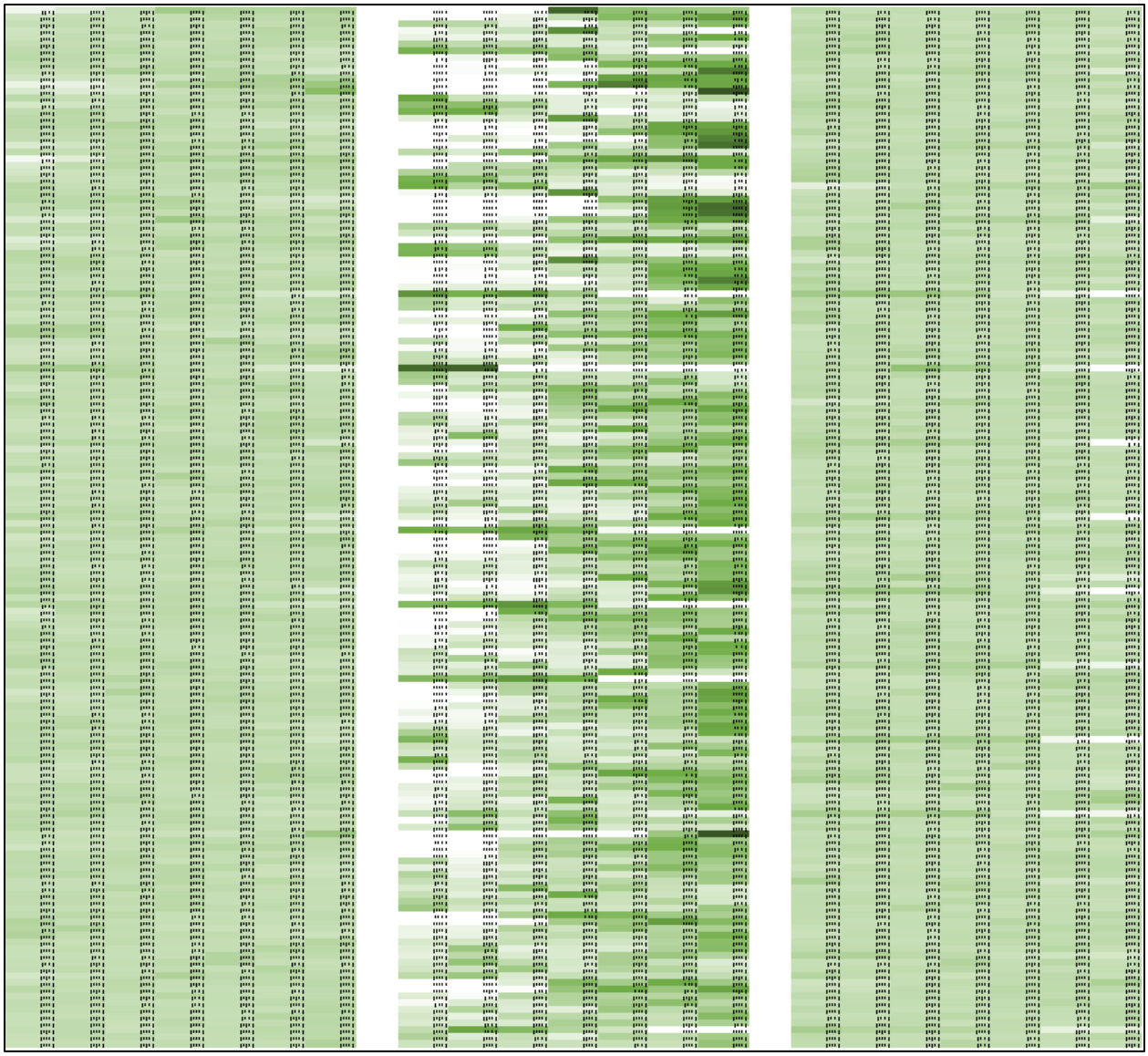

Figure 4.10 Expected Policies for reward functions shown in Figure 4.8

We shall now look at the 2nd section's reward function and try to interpret it better. Figure 4.11 focuses on the middle section's policy.

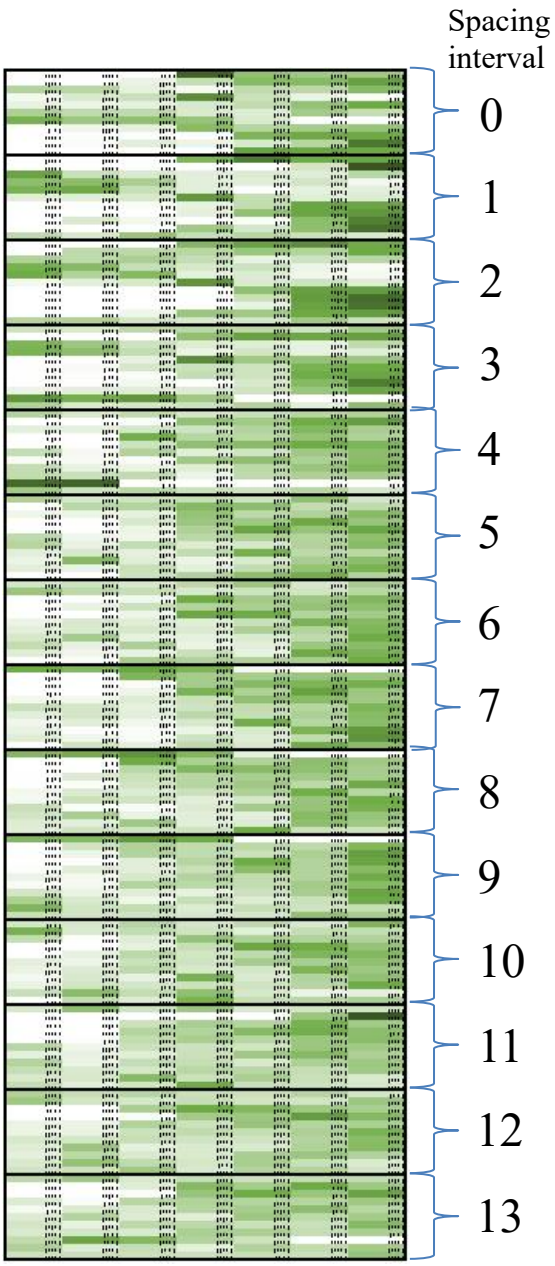

Figure 4.11 Reward Function of Merge section

In Figure 4.11, each of the 13 partitions of the reward function correspond to a spacing interval which means the spacing is the same for each partition. However, the relative velocity decreases from 45 ft/s to -45 ft/s from top to bottom within a partition. In each section, the top half corresponding to positive relative velocity (car A is faster than car C) has distributions skewed towards braking actions, while the bottom half corresponding to negative relative velocity (car C

is faster than car A) has distributions skewed towards accelerating actions, with a few exceptions. Also, we see a fair amount of states where car A chooses the counter intuitive action of accelerating when clearly, it seems to be a risky choice.

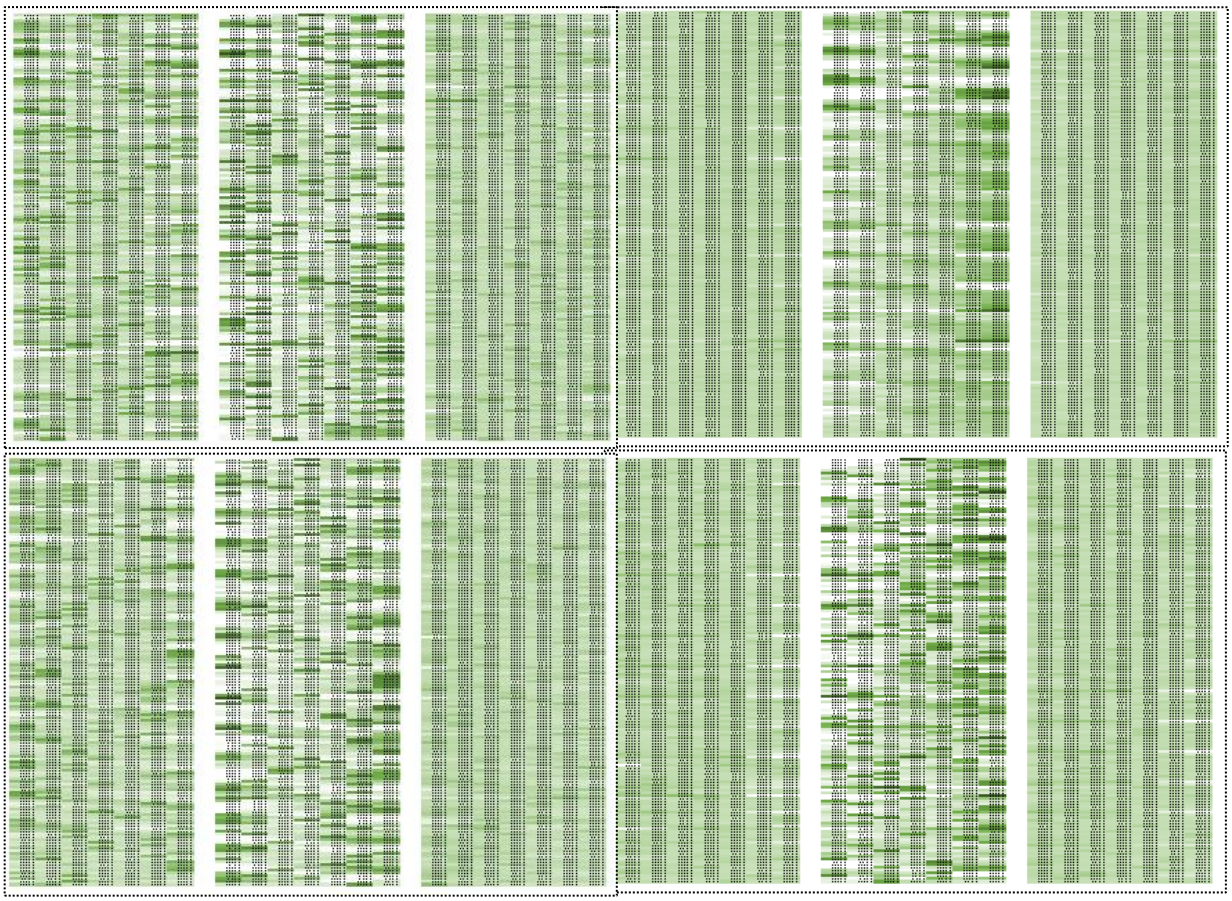

Figure 4.12 More Expected policies from runs of 1M samples

Clearly, we can see some variation in the results from different runs. The main factor for this variation is probably the lack of information for a number of states in the trajectories as we have seen before. In the absence of this information, the degeneracy issue causes multiple reward functions to be able to explain the demonstrated behaviors equally well, which in turn leads to different policies based on the expected c parameter values.

CHAPTER 5

CONCLUSION AND FUTURE WORK

The different results obtained from LWBMIRL clearly show that there is a difference in driver policies in the pre-merge, merge and post-merge sections. Thus, if we consider the task of merging as a combination of the three aforementioned tasks, then our hypothesis that the behavior of car A changes depending on which task it is performing, is verified.

However, looking at the variance in the set of reward functions and, to some extent, the set of policies obtained from different runs, we might say that there are multiple reward function sets that explain car A's behavior change. A good reason for multiple reward functions supporting the demonstrated behavior of car A is the previously discussed degeneracy issue.

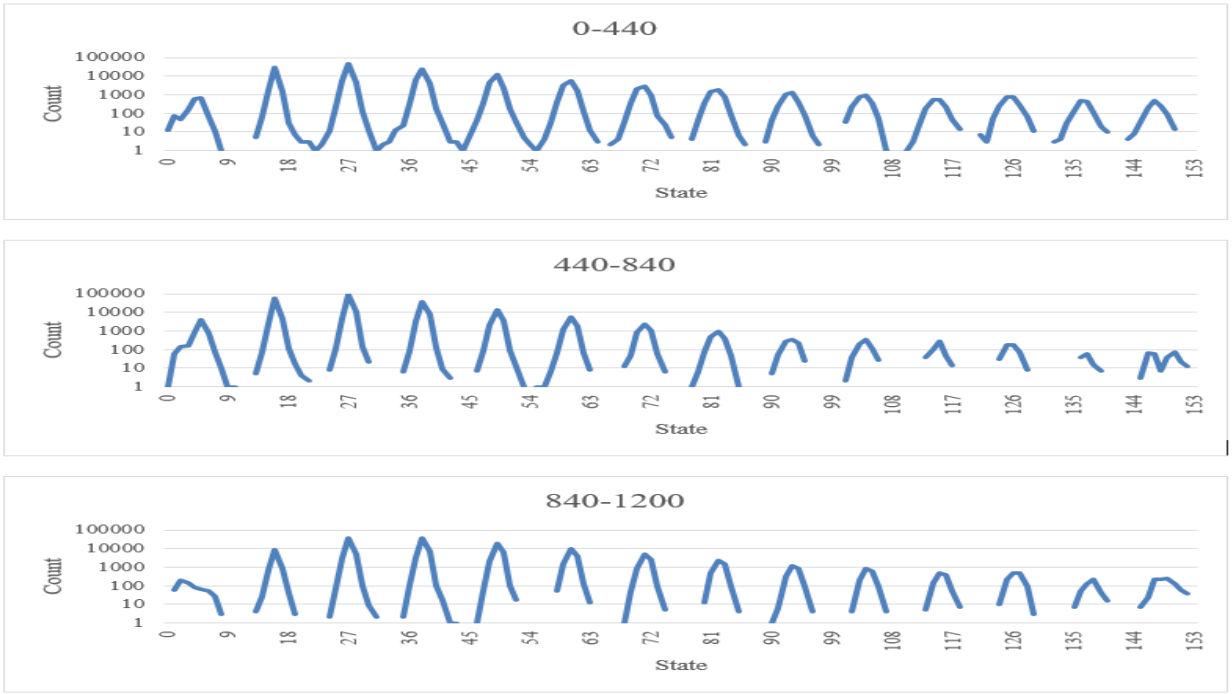

Figure 5.1 State Count in each section on logarithmic scale

In the absence of enough data for actions performed in a number of states, this degeneracy is further exacerbated. Figure 5.1 shows the state-action frequency for each of the three sections. The discontinuity in the graphs correspond to states with zero count on account of the logarithmic scale of the counts axis. Thus, we could say that our method is not good at finding a maximum entropy posterior when the data lacks enough information about all states in its trajectories.

Another reason for our method not being able to arrive at a converged posterior could potentially be due to the choice of such a prior whose support is almost disjoint from the posterior. Due to this, most samples are likely to have lower likelihood weights and thus the posterior may not correspond to a true posterior. We have proposed an improvement for this later in this section.

However, our method should be able to perform better by making some improvements. We propose the following future enhancements:

- Implicit sampling[16]

  Implicit sampling is an importance sampling technique aimed at generating implicit samples that receive large posterior probability instead of generating a large number of prior samples which rarely fall in the high posterior region. The main idea here is to construct a data-informed prior which significantly overlaps with the posterior so that the prior places higher importance to those samples with potentially high likelihood.

- Eliminate c-parameter

  In our algorithm, we assume that the expert trajectory may not be optimal and hope to surpass the expert by using a parameter which defines a distribution over the optimal policy based on the expert reward inferred. However, the addition of this extra branch of random variables puts a significant penalty on the effectiveness of the prior. However, we could

remove the c parameter and still compute stochastic policies directly from the reward functions. Based on Eq 1.3, we can define a distribution over actions given a state as

$$\pi^*(a \mid s) \propto \sum_{s'} T(s, a, s') * V(s') \quad \text{... Eq 5.1}$$

Where the normalization is done with respect to all actions giving,

$$\pi^*(a \mid s) = \frac{\sum_{s'} T(s, a, s') * V(s')}{\sum_{a'} \sum_{s'} T(s, a', s') * V(s')} \quad \text{... Eq 5.2}$$

Thus, we can still compute stochastic policies to evaluate the likelihood without using the c parameter. Of course, this method would learn the expert's reward function and generate a policy assuming it is optimal.

- Better state space

  Our state space was chosen with the features having mostly equal sized intervals. This however may not be able to capture the fine details especially for small spacing values. A better state space with finer granularity in the small spacing region should effectively highlight the differences in the true conditions perceived by the car A driver.

REFERENCES

[1] Jochem, T., Pomerleau, D., Kumar, B. and Armstrong, J., 1995, September. PANS: A portable navigation platform. In *Intelligent Vehicles' 95 Symposium., Proceedings of the* (pp. 107-112). IEEE.

[2] Das, I., 2016, August. Inverse reinforcement learning of risk-sensitive utility. (Masters' Thesis, University of Georgia Athens)

[3] Russell, S.J., Norvig, P. & Davis, E., 2010. *Artificial intelligence: a modern approach* 3rd ed.(Vol 2), Boston: Pearson.

[4] Russell, S., 1998, July. Learning agents for uncertain environments. In *Proceedings of the eleventh annual conference on Computational learning theory* (pp. 101-103). ACM.

[5] Ng, A.Y. and Russell, S.J., 2000, June. Algorithms for inverse reinforcement learning. In *Icml* (pp. 663-670).

[6] Ziebart, B.D., Maas, A.L., Bagnell, J.A. and Dey, A.K., 2008, July. Maximum Entropy Inverse Reinforcement Learning. In *AAAI* (Vol. 8, pp. 1433-1438).

[7] Ramachandran, D. and Amir, E., 2007. Bayesian inverse reinforcement learning. *Urbana*, *51*(61801), pp.1-4.

[8] Rothkopf, C. and Dimitrakakis, C., 2011. Preference elicitation and inverse reinforcement learning. *Machine Learning and Knowledge Discovery in Databases*, pp.34-48.

[9] Dimitrakakis, C. and Rothkopf, C.A., 2011, September. Bayesian multitask inverse reinforcement learning. In *European Workshop on Reinforcement Learning* (pp. 273-284). Springer Berlin Heidelberg.

[10] Babes, M., Marivate, V., Subramanian, K. and Littman, M.L., 2011. Apprenticeship learning about multiple intentions. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)* (pp. 897-904).

[11] Choi, J. and Kim, K.E., 2012. Nonparametric Bayesian inverse reinforcement learning for multiple reward functions. In *Advances in Neural Information Processing Systems* (pp. 305-313).

[12] Wilson, A., Fern, A., Ray, S. and Tadepalli, P., 2007, June. Multi-task reinforcement learning: a hierarchical Bayesian approach. In *Proceedings of the 24th international conference on Machine learning* (pp. 1015-1022). ACM.

[13] US Department of Transportation, 2006, December. Fact Sheet-Interstate 80 Freeway Dataset, *Federal HighWay Administration (FHWA).*

[14] Geweke, J., 1989. Bayesian inference in econometric models using Monte Carlo integration. *Econometrica: Journal of the Econometric Society*, pp.1317-1339.

[15] Thrun, S., Burgard, W. and Fox, D., 2005. *Probabilistic robotics*. MIT press.

[16] Morzfeld, M., Tu, X., Wilkening, J. and Chorin, A., 2015. Parameter estimation by implicit sampling. *Communications in Applied Mathematics and Computational Science*, *10*(2), pp.205-225.

APPENDICES

A. Importance Sampling Perspective of Likelihood Weighting

Let $f(x) = (\rho_1, \rho_2, \rho_3, c_1, c_2, c_3)$ be a random variable representing rewards and policy parameters given evidence D that we wish to compute. Let P(X) represent the true posterior of f(x). The expected value of f(x) with respect to P(X) is simply given as

$$E_P[f(x)] = \int f(x)\, P(x)dx$$

$$= \frac{1}{K}\sum_{k=1}^{K} f(x[k]) \qquad \text{... Monte Carlo approximation}$$

But we cannot sample from P(X) as we do not know the true posterior and hence, we use a proposal distribution Q(X) to generate samples from and with correction applied for sampling from the proposal, we estimate the expected value with respect to Q(X) as

$$E_Q[f(x)] = \int f(x)\, \frac{P(x)}{Q(x)} Q(x)dx$$

$$= \frac{1}{K}\sum_{k=1}^{K} f(x[k]) \frac{P(x[k])}{Q(x[k])}$$

The problem is still that we do not have a way to compute P(x). However, we can compute the unnormalized posterior P'(X) with respect to the samples drawn. Hence, we have P'(X) = Z P(X), where $Z = \sum_x P'(x)$ is the normalization constant. Now let us introduce

$$w(X) = \frac{P'(X)}{Q(X)}$$

whose expected value with respect to Q(X) is simply Z. This is shown as follows:

$$E_Q[w(x)] = \sum_x w(x)\ Q(x) = \sum_x \frac{P'(x)}{Q(x)}\ Q(x) = \sum_x P'(x) = Z$$

Now, we rewrite $E_P[f(x)]$ as

$$E_P[f(x)] = \sum_x P(x)\ f(x)$$

$$= \sum_x \frac{P(x)}{Q(x)}\ f(x)Q(x)$$

$$= \sum_x \frac{w(x)}{Z}\ f(x)Q(x)$$

$$= \frac{\sum_x (w(x)\, f(x))\ Q(x)}{E_Q[w(x)]}$$

$$= \frac{E_Q[w(x)\, f(x)]}{E_Q[w(x)]}$$

which can be computed by Monte Carlo approximation with K samples from Q(X) as

$$\widehat{E_P}[f(x)] = \frac{\sum_{k=1}^{K} f(x[k])\ w(x[k])}{\sum_{k=1}^{K} w(x[k])}$$

The likelihood weight w(x) makes up for the difference between P(x) and Q(x).