# Bayesian Networks in Lane Change Maneuver Prediction

**Grabić, Ivan**




UNIVERSITY OF ZAGREB
FACULTY OF MECHANICAL ENGINEERING AND NAVAL ARCHITECTURE

# MASTER'S THESIS

**Ivan Grabić**

Zagreb, 2020

UNIVERSITY OF ZAGREB
FACULTY OF MECHANICAL ENGINEERING AND NAVAL ARCHITECTURE

# MASTER'S THESIS

## BAYESIAN NETWORKS IN LANE CHANGE MANEUVER PREDICTION

Mentor:
Doc. Dr. sc. Petar Ćurković

Student:
Ivan Grabić

Zagreb, 2020

The support of the Erasmus+ program is gratefully acknowledged.

This thesis was created in collaboration with Audi AG.

I would like to give my sincere thanks to my supervisor at Audi, Mrs. Toshika Srivastava, who gave me the opportunity to write my thesis in such a creative and professional environment. Her pieces of advice, patience, and ideas have been valuable contributions in performing research and writing this thesis.
I would also like to extend my gratitude to professor Petar Ćurković, my thesis supervisor, for making this collaboration easy and for his valuable comments and suggestions.
My great appreciation goes to all of my other colleagues at Audi, especially to my colleague and friend Igor Katulić. My thanks to all my university colleagues and professors who made my stay in Germany possible.
Finally, I would like to express love and gratitude to my family - father Ante, mother Irena, sister Anamarija, and brother Tomislav, as well as to my girlfriend Anamaria and lifelong friends for their love and support.

Ivan Grabić

I hereby declare that I have made this thesis independently using the knowledge acquired during my studies and the cited references.

Izjavljujem da sam ovaj rad izradio samostalno koristeći znanja stečena tijekom studija i navedenu literaturu.

Ivan Grabić

SVEUČILIŠTE U ZAGREBU
**FAKULTET STROJARSTVA I BRODOGRADNJE**
Središnje povjerenstvo za završne i diplomske ispite
Povjerenstvo za diplomske ispite studija strojarstva za smjerove:
procesno-energetski, konstrukcijski, brodostrojarski i inženjersko modeliranje i računalne simulacije

| Sveučilište u Zagrebu<br>Fakultet strojarstva i brodogradnje | |
|---|---|
| Datum: | Prilog: |
| Klasa: 602 - 04 / 20 - 6 / 3 | |
| Ur. broj: 15 - 1703 - 20 - | |

# DIPLOMSKI ZADATAK

Student: **Ivan Grabić** Mat. br.: 0035196505

Naslov rada na hrvatskom jeziku: **Primjena Bayesovih mreža za predviđanje manevra promjene prometne trake**

Naslov rada na engleskom jeziku: **Bayesian networks in lane change maneuver prediction**

Opis zadatka:

Probabilistic machine learning methods are recently getting increased attention both in research and production community. The raise of autonomous vehicles domain is boosting different aspects of Artificial intelligence based methods and their implementation. Available computational power enables the real time usage of such methods to: recognize objects; plan trajectories, evaluate driver's condition; optimize energy consumption, to name a few.
In this thesis, a Bayesian network should be developed in order to enable prediction of the lane change maneuver. In particular, the following should be addressed:

- Explore the theoretical background of Bayesian networks and their applicability for the problem described
- Explore the Highway Drone Dataset (HighD), develop appropriate algorithmic procedures to clean the data, and prepare it for importing into the Bayesian network
- Propose a topology of the Bayesian network to describe the problem realistically
- Use a probabilistic programming language (PPL) based on Python-Pyro to implement the Bayesian network and learn the parameters
- Choose adequate inference algorithm to fine tune learning and predicting procedures
- Perform statistical analysis of the prediction reliability for different parameters used, and use these data to evaluate the model and give a critical review.

Please list the literature sources used, and indicate any help recieved in writing of the thesis.

Zadatak zadan: 24. rujna 2020.

Datum predaje rada: 26. studenoga 2020.

Predviđeni datum obrane: 30.11. – 4.12.2020.

Zadatak zadao:

Doc. dr. sc. Petar Ćurković

Predsjednica Povjerenstva:

Prof. dr. sc. Tanja Jurčević Lulić

# Contents

# List of Figures

# List of Tables

# List of symbols

| | |
|---|---|
| $X, Y$ | random variable |
| $x, y$ | events |
| $P(x)$ | probability of an event $x$ |
| $P(x\|y)$ | probability of an event $x$, given $y$ |
| $\mathbb{E}$ | expectation |
| $H$ | Shanon entropy |
| $L$ | evidence lower bound |
| $KL$ | Kullback-Leiber divergence |
| $q(z)$ | variational distribution |
| $\rho$ | learning rate |
| $\lambda$ | variational parameter |
| S | number of samples |
| $\mathcal{N}$ | normal distribution |
| $\mathcal{B}$ | Bernoulli distribution |
| $\mu$ | mean |
| $\sigma$ | standard deviation |
| $\theta$ | parameter in a network |

# SAŽETAK

Razvoj autonomnih vozila izazovan je zadatak. Jedan od razloga tome je što je ljudsko ponašanje nepredvidljivo. Drugi je razlog taj što je ovaj problem u visoko rizičnom okruženju. Većina prometnih nesreća dogodila se zbog ljudske pogreške. Stoga se može zaključiti da će autonomna vozila vožnju učiniti sigurnijom.

Jedna vrsta nesreće dogodi se kada jedan sudionik promijeni traku, a drugi sudionik to ne primijeti. Ako bi sustav mogao predvidjeti promjenu trake i na vrijeme upozoriti vozača, mogao bi spriječiti nesreću. Metode dubokog učenja su najsuvremeniji pristup problemima predvianja. Neuronske su mreže, meutim, poznate kao crne kutije. To je razlog što nisu u potpunosti prikladne za rizične domene poput prometnog okruženja.

Ovaj rad će zauzeti alternativni pristup predvianju manevara promjene trake. Taj se pristup naziva Bayesovim ili probabilističkim strojnim učenjem. Dvije su glavne prednosti ovog pristupa. Prva je interpretabilnost probabilističkih modela, a druga dobro prikazivanje nesigurnosti.

Stvorit ćemo Bayesovu mrežu za predvianje manevara promjene trake sudionika u prometu. Razmotrit ćemo "Highway Drone Dataset" (HighD) i pokazati zaključke. Od ovog skupa podataka stvorit ćemo set podataka za učenje i testiranje. Za stvaranje modela poslužit ćemo se vjerojatnosnim programskim jezikom zvanim pyro. Model ćemo trenirati na setu podataka za učenje koristeći algoritam nazvan "Black Box Variational Inference".

Nakon učenja, model se evaluira te su prikazane metričke vrijednosti. Vrijeme predvianja prikladno je za provedbu u stvarnom vremenu. Moć predvianja usporediva je s drugim probabilističkim pristupima, ali je gora od modela dubokog učenja.

Ključne riječi: Probabilističko strojno učenje, probabilističko programiranje, predvianje manevra promjene trake, Bayesove mreže, autonomna vožnja.

# SUMMARY

Developing autonomous vehicles is a challenging task. One of the reasons for this is that human behavior is unpredictable. Another reason is that this problem is in a high-risk environment. Most traffic accidents are due to human error. Thus a conclusion can be made that autonomous vehicles will make driving safer.

One type of accident happens when one participant changes the lane and the other participant doesn't notice it. If a system could predict lane change and alert the driver in time it could prevent an accident. Deep learning methods are state of the art approach to prediction problems. Neural networks are, however, known as black-box models. This is the reason they are not fully suitable for high-risk domains such as traffic environment.

This thesis will take an alternative approach to lane-change maneuver prediction. This approach is called Bayesian or probabilistic machine learning. There are two main benefits to this approach. First is interpretability of probabilistic models and second is good uncertainty representation.

We will create a Bayesian network for predicting lane change maneuvers of traffic participants. We will look at Highway Drone Dataset (HighD) and show conclusions. From this dataset, we will create a training and test dataset. To create a model we will use a probabilistic programming language called pyro. We will train the model on a training set using an algorithm called Black Box Variational Inference.

After the training, the model is evaluated and evaluation metrics are reported. The inference time is appropriate for real-time implementation. Prediction power is comparable with other probabilistic approaches but worse than deep learning models.

Keywords: Probabilistic machine learning, probabilistic programming, lane-change maneuver prediction, Bayesian networks, autonomous driving

# 1 Introduction

## 1.1 Motivation

One of the strongest motivations in autonomous driving development is decreasing accidents. The human factor is the number one reason for car accidents. In 95% of accidents human factor is involved and 75% of accidents are due to human error alone. The countires with most accidents in Europe: Bulgaria, Romania, and Croatia have more than 70 fatalities per million inhabitants. In addition to lost lives and devastating effect on affected families, the cost of road fatalities is estimated to be 100 billion euros a year for European Union [1].

On the way to fully automated highway researchers are developing Advanced Driver Assistance Systems (ADAS). ADAS is a collection of systems and subsystems which are meant to enable comfortable and safe driving to a driver. From 2001. and 2017. the number of accidents decreased 57,5% [2]. A lot of that progress is thanks to the advancement in ADAS functions [2]. Besides technology, a lot of regulations (for example regulating drinking and driving) helped in reducing the number of accidents. Figure 1 shows number of fatal accidents from 1950. in Germany and it's correlation with state regulations

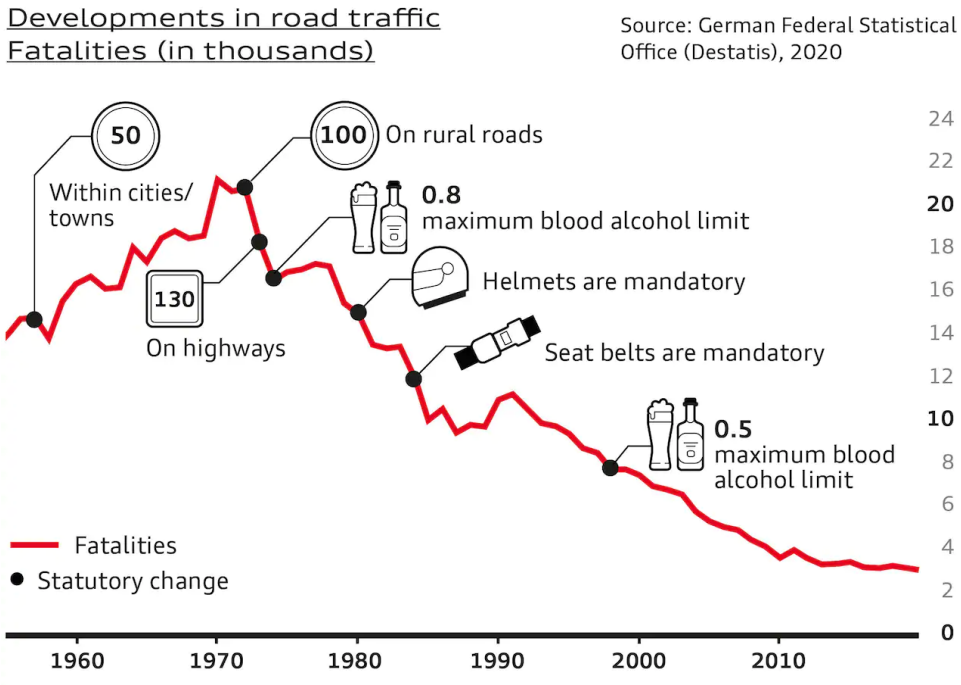


**Figure 1. Fatal accidents in Germany from 1950. to 2020. Source: German Federal Statistical Office (Destatis), 2020**

Study [3] shows that 17% of all crashes were due to lane change. They show that the injuries are more severe in lane change crashes in comparison with other types of crashes

(Figure 2). The number one reason for accidents is a distraction. Generally, 57,2% of accidents are due to distraction, and 21,2% due to mobile phone use only. The study [4] shows that 85% of drivers use a turn signal when performing planned lane change. When drivers perform unplanned lane change, due to possible forward crash, 24% use a turn signal. One can assume that it is hard to predict the intentions of other participants while driving. Even without possible distractions sudden lane change without turn signal can surprise drivers and lead to a crash.

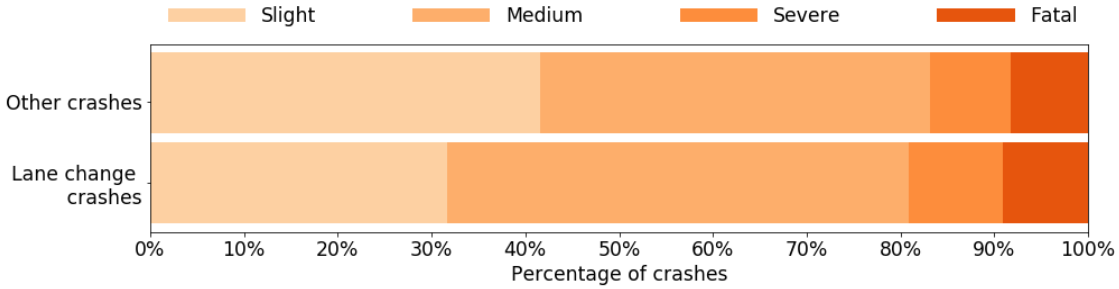


**Figure 2. Visualisation of table 1 (Statistics of injury-severity levels) from [3].**

The injuries from lane change related crashes is more severe than injuries from other type of crashes.

## 1.2 Problem statement

A system that could predict these sudden lane changes would be helpful in drivers' decision making. As a consequence drivers would have a safer experience. In the most common type of lane change related accident, the lane change is intentional. One vehicle intentionally changes lane and sideswipes vehicle in another lane, or being sideswiped by another vehicle. This type of accidents accounts for more than 38% of crashes [4]. If some system can infer that intention based on observations it could help avoid accidents.

We can say that such a system should reason about traffic participant intentions. How do we develop systems that can reason? There are various approaches to solving this problem. One opinion divides these approaches into two main clusters. That opinion is that random variables and probability calculus are the most important tools in building reasoning machines [5]. This opinion forms field of Bayesian or Probabilistic machine learning. The opposite opinion forms a field of Deep learning.

To build this system you need to have a model of a traffic situation as a part of a system. Model building is a common practice in science and engineering. We can look at a model as an artificial construction of a system we want to understand. This system should respond in the same way as the original system. With advances in computation numerical models and simulations replaced physical models used in the past. If a simulation models random

phenomena with pseudorandom generators we say that model is stochastic. In the context of this thesis, the term model means the stochastic simulator and the values it produces. We can see that models range from interpretable simulators used in engineering to statistic and machine learning models that are less interpretable but have good predicting power.

All models have parameters. We should select such parameters to fit the model so that model produces the observable data. If the model has a small number of parameters it could be fitted mannually. For bigger models with many parameters, algorithms exist that can automate model fitting. The automated model fitting for a model is called learning.

Systems that are used in traffic scenarios are considered to be in the high-risk domain. This means that there are high prices to pay for the wrong decisions In contrast, there are models developed for low-risk domains as games or image recognition. In these kinds of models, the wrong prediction can be bothersome but not dangerous.

We want the model to be useful. This means primarily that predictions of a model are useful. But for high-risk domains, the interpretability of a model is also useful. The model should be transparent so that we can trust it. There are legal arguments for having interpretable models. With General Data Protection Regulation taking effect in 2018,European Union mandates a "right to an explanation". This means that users can legally ask for an explanation of an automated system decision. There is a tradeoff between interpretability and predictive power, but there are ways to increase interpretability with an insignificant decrease in predictive power.

Uncertainty representation is also a useful property of a model. A favorable system should integrate information about uncertainty into a decision-making process. ADAS system can acknowledge uncertainty on different levels, from sensor uncertainty to prediction uncertainty. Uncertainty can propagate from sensors to high-level decision making. The system should rely on model uncertainty to adjust the decision-making process. This way system can avoid unintended behavior.

Let us introduce a general information flow of the ADAS system to understand uncertainty propagation (Figure 3). Information flows from low-level sensor systems to high-level behavior generation. In the meantime, information is passed through different process units. For different ADAS features different stages are not executed or not as relevant. The feature extraction step gets higher requirements as ADAS features get more complex.

Only recently probabilistic approaches have become a mainstream approach in artificial intelligence, robotics, and machine learning [7]. Probabilistic approaches are often used in areas where uncertainty and interpretability are requirements.

Almost every machine learning task is learning about latent variables given some observations. We define the model as good if it can predict some data after being trained

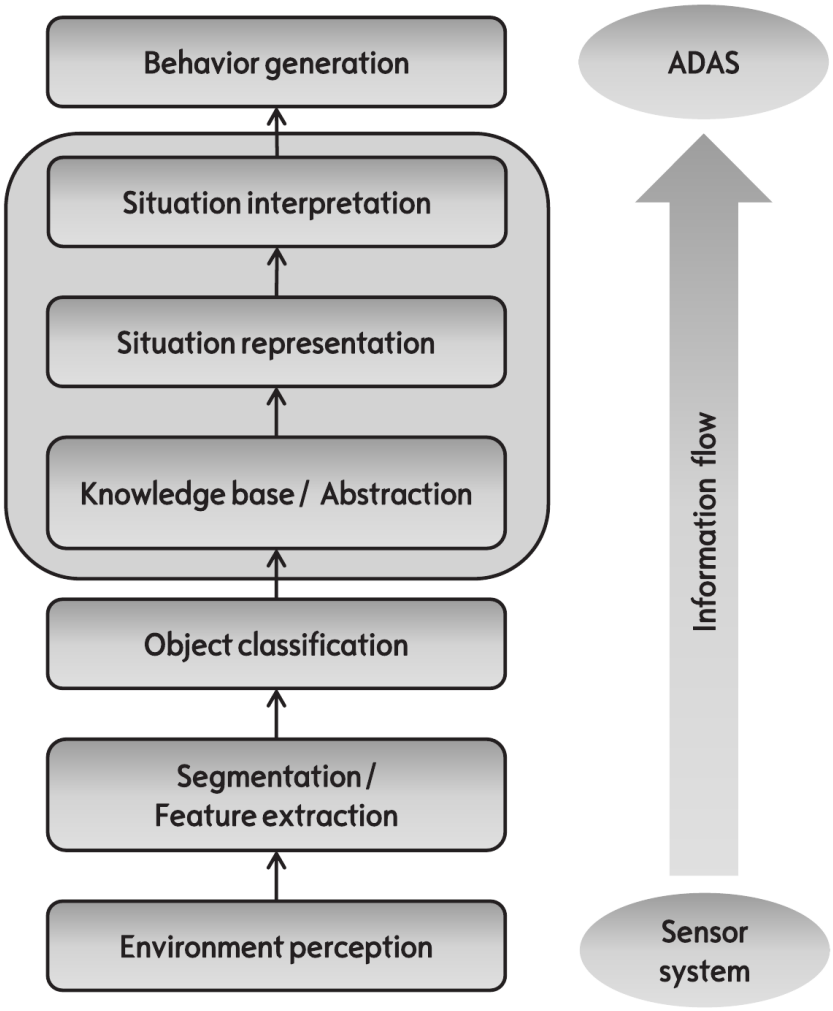


**Figure 3. Information flow of ADAS [6]**

Information flows from sensors and to the behavior generation module. It is important to acknowledge uncertainty for every step so that it can influence behavior generation.

on observed data. Different levels of uncertainty can be presented in a machine learning task. On the lowest level, it can represent data uncertainty. On higher levels model can represent uncertainty about its parameters. The highest level is uncertainty about the model structure itself.

In the same way, calculus is useful for representing and manipulating with the rate of change, probability theory is used for representing uncertainty. We represent unobserved quantities in model with probability distributions. We use rules of probability calculus to infer unobserved quantities from observed data. Learning is when probability distribution is transformed form prior (before observing data) to posterior (after observing data). Applying probability theory to learning is called Bayesian learning.

One of the main advantages of probability calculus is the possibility of joining simple

probability distributions in complex models. Creating a graphical model is the way of representing this joined distribution over some variables. Graphical models are graphs where nodes represent random variables, edges represent dependencies, and with parameters that quantify the dependencies. We call directed graphical model a Bayesian network and undirected Markov network. Compositionality of these models gives them interpretability [7].

## 1.3 Related work

Some approaches use Bayesian networks and similar models for lane change maneuver analysis. The motivation for this approach is the interpretability of the bayesian network, and it's a natural representation of uncertainty. The Bayesian network explicitly shows dependencies between random variables. You can encode expert knowledge in the bayesian network. The parameters of Bayesian networks are probabilities which are a natural representation of uncertainty.

Three types of features used for maneuver predictions are [8]:

- physical state of the vehicle,
- road structures related features,
- traffic interaction related features.

Physics-based features refer to the kinematic state of the vehicle. Those are features like position, velocity, and acceleration of vehicles. Road structure features are referring to road topology, road signs, and traffic rules. Many approaches combine these types of features. When using interaction aware features you consider dependencies between vehicles.

Not many approaches considered interaction-based features before 2014 [8]. The physics-based models were used like: constant velocity, constant acceleration, constant turn rate or combination. The intelligent driver model is a model that explicitly represents the relationship between intentions on intersections and velocity and acceleration. These models are efficient for short time intervals. The disadvantage of these models is that they are hard to adapt to different scenarios.

Recently researchers started to model maneuver prediction using interaction based features [8]. These features represent how is motion influenced by vehicle interactions. With deep learning methods gaining traction some researchers used it for maneuver-prediction tasks. In [9] researches use Convolutional Neural Network for lane change maneuver prediction. Another approach is using Long Short Term Memory (LSTM) [10] neural network. LSTM is a recurrent neural network appropriate for time series data.

Deep learning models implicitly model interaction related features. Neural networks don't have an intuitive prediction process. They are notably known as black boxes because of their lack of interpretability. Their non-interpretability is related to their architecture. When inferring prediction form input data, data is propagated trough many layers of nonlinear transformations. These nonlinear transformations make it hard to interpret results.

For explicitly using interaction features researchers are using Bayesian networks and their extensions. The approach in [11] is using object-oriented Bayesian Networks (OOBN) for lane change maneuver prediction. This approach represents network segments with so-called instance nodes. The output of the network is a single node with values: Following Object, Following Lane, Cut in/out. Paper [12] has an interesting approach because it models all traffic participants together instead of each of them separately. The model, however, requires a lot of computational power because of its size.

Authors of [8] proposed maneuver prediction approach based on Dynamic Bayesian networks. Authors report the 3,75 seconds prediction time an 80% F1 score. This network, however uses only discrete nodes.

In [13] Bayesian network is used for lane change maneuver prediction in extra urban traffic scenarios. Extra urban traffic scenarios are situations which are outside the urban streets like highways and rural roads. Urban streets require more constraints and are more complex. They model Bayesian network with discrete nodes and output is decision node with values: Keep Lane, Change Lane Left/Right.

In [14] Bayesian network is created with three nodes for right turn assist. Nodes in Bayesian networks are intention I, hypotesys H, and observation O. The task is to calculate $P(I|O)$. They use Intellignet driver model to drive hypoteses H.

## 1.4 Thesis overview

This thesis will show the development of a simple and interpretable probabilistic model. This model will have good uncertainty representation and predictive power. We will create such a model using probabilistic programming. I will present a theoretical background for Bayesian networks. Also, show a history of developing inference algorithms. After that, I will show that variational inference is a promising inference algorithm. I will present the meaning of probabilistic programming in Bayesian machine learning. In section 3 I will explore data and describe the process of data preparation. In 4 we will see model structure and process of training generative model, and inferring intention from the trained model. The model will be evaluated and results will be presented in 5. Finally, in section 6 the thesis method, result, and limitations are going to be discussed. Also, the future work and extensions will be proposed in that section.

# 2 Bayesian Machine Learning

Alan Turing gave a basic notion of what an Artificial Intelligence (AI) is. In [15] he proposed a test which he called "imitation game" and which is today known as the Turing test. An artificial intelligent agent would be, according to his proposition, agent that could fool ordinary human about whether the human communicates with a human or a machine. This level of intelligence is something researches are striving for in AI system development.

Intelligent machines should reason logically. This is a widespread fact in the AI community. A big portion of the AI community also thinks that such machines should deal with uncertainty. However, only a smaller part believes that intelligent machines should also reason probabilistically [16]. The field of machine learning that was born from a belief that AI should reason probabilistically is called Bayesian (also called Probabilistic) Machine Learning.

We can say that Bayesianism is a philosophy that asserts that: " in order to understand human opinion as it ought to be, constrained by ignorance and uncertainty, the probability calculus is the single most important tool for representing appropriate strengths of belief." [16]

## 2.1 Probability calculus and machine learning

Let us introduce basic probability calculus. Two rules underlie probability theory:
the sum rule:

$$P(x) = \sum_{y} P(x, y), \tag{2.1}$$

and the product rule:

$$P(x, y) = P(x)P(y|x). \tag{2.2}$$

Here $x$ and $y$ are some observed uncertain quantities. For example, $x$ can represent a driver's intention on the intersection and $y$ can represent a planned destination. Both $x$ and $y$ are values taken from sets $X$ or $Y$ respectively. Here $X$ is for example set of values $forwad, left$. $P(x)$ is then probability of event $x$. $P(x, y)$ is the joint probability of observed events. Finally, $P(y|x)$ is conditional probability. This represents a probability of $y$ given that we know the value of $x$. We say that the probability of $y$ is conditioned on observing the value of $x$ and call this process conditioning. These probabilities can be interpreted as the frequency of observing that value, or as a subjective degree of belief, that event will take place.

These two interpretations of probabilities create two branches of statistics: Bayesian statistics and Frequentist statistics. Frequentist believes that probability is a fundamental property of a random physical system. A frequentist will associate the probability of dice

rolling to a specific number with a frequency of that event imagining that he will roll the dice many times. Bayesinism is an extension of that interpretation. It states that probability is subjective partial belief. One can have a belief in some event even with that event untied to a physical process that is repeatable many times. We can talk about what is the probability that every car in the world will be autonomous in the next ten years. Probability isn't fundamental physical property in this case. We can't, even in principle, repeat this experiment many times. Most frequentists will, therefore, consider this probability meaningless. Bayesianist can talk about this probability and its relation to evidence for and against it.

From (2.1) and (2.2) we can derive Bayes theorem:

$$P(y|x) = \frac{P(x|y)P(y)}{P(x)}. \tag{2.3}$$

To apply probabilistic calculus to machine learning we can replace $x$ and $y$ in 2.3 with $D$ and $\theta$ respectively. Here $D$ represents data on which you want your model to learn, and $\theta$ are parameters of your model.

Additionally, you condition all terms on $m$, which represents model architecture. So the expression 2.3 becomes:

$$P(\theta|D, m) = \frac{P(D|\theta, m)P(\theta|m)}{P(D|m)}. \tag{2.4}$$

.

In (2.4) $P(D|\theta, m)$ is called likelihood of parameters $\theta$ of a model $m$. $P(\theta|m)$ is called a prior probability of parameters or prior. It represents the probability of parameters before training. $P(\theta|D, m)$ is a posterior probability of parameters, after considering data.

## 2.2 Bayesian network basics

Probabilistic graphical models (PGMs) are a powerful framework for representing uncertain systems. We can look at PGMs as probability distributions joined together. This compositionality makes them more interpretable. They are used in many applications of machine learning, computer vision, natural language processing, etc. We represent them as graphs where nodes represent random variables and arcs represent conditional dependencies. In this approach, I will use a Bayesian network, which is a type of PGMs to solve maneuver prediction problem.

Bayesian network (BN) is a directed acyclic graph in which each edge corresponds to conditional dependency and each node corresponds to a unique variable. [17] By definition, BN brings together graph theory and probability calculus. Let us introduce a

mathematical representation of a Bayesian network.

A graph $G = G(V, E)$ consists of a set of objects V called vertices (or nodes) and another set of objects E called edges. The directed graph additionally consists of mapping that maps every edge to a set of ordered nodes [18]. In other words, the map gives direction to edges, and we will call directed edges arcs. The term acyclic refers to a constraint that there can't be any directed cycles in the graph. That means that you can't return to a node by following directed arcs [16]. A simple directed acyclic graph would look like this.


**Figure 4. Simple Directed Acyclic Graph**

Directed acyclic graph consists of nodes and directed edges. The fact that is directed means that you can not return to a node just by following directed edges.

A random variable is a variable which reports the outcome of some measurement process [16]. A random variable $X$ has multiple states that it can take.

$$X = \{x_1, x_2, ..., x_n\} \tag{2.5}$$

A variable can be discrete or continuous. The example of a discrete variable is a simple boolean variable with True and False states. On the other hand, we have a continuous variable, which ranges over real numbers, like for example speed or temperature. The Bayesian network is primarily oriented to handling discrete variables [16]. In most cases, continuous variables can be discretized. Sometimes with discretization of continuous variables we loose some information. However, continuous variables can be also used, and are used in this thesis. Hybrid Bayesian network contain discrete and continuous variables.

When modeling an uncertain problem, one should consider which variables are of interest. These variables condition what nodes will network have and what values will those nodes take. One constraint on values is that they must be mutually exclusive and exhaustive [16]. That means that the variable can take just one value at the time. After the nodes are being defined, one should define qualitative relationships between the nodes. We already mentioned that relationships can be represented by arcs. In particular, two nodes should be connected by an arc if one affects or causes the other [16]. When all relationships are presented, we have structure (or topology) of a network.

When talking about networks we are going to borrow terminology from [16], which uses the family metaphor. If there is an arc from node A to node B, we say that A is

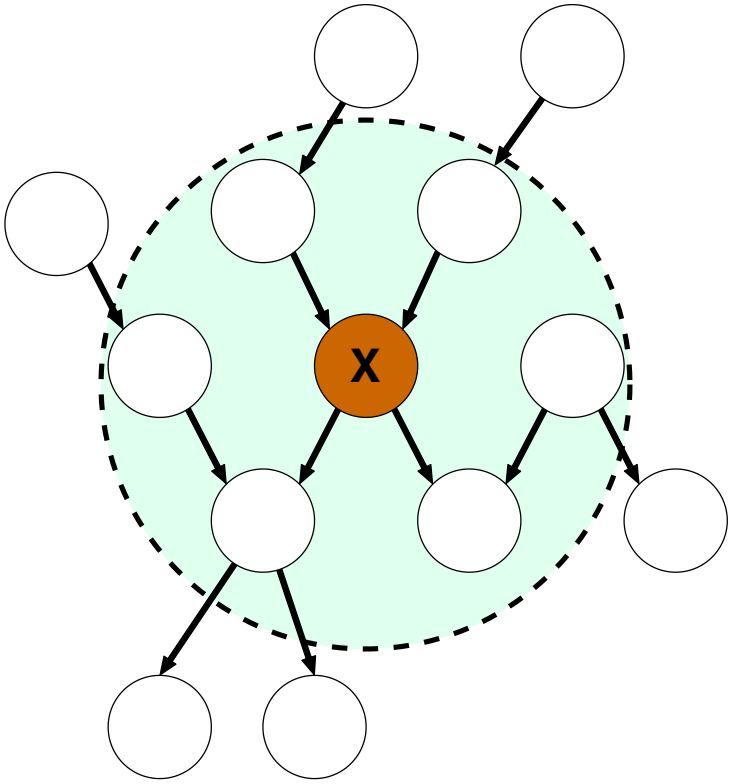


**Figure 5. Markov blanket of a node.**

The markov blanket consists of a node, its parents and its child parents. In the picture the markov blanket of node X are every note in shaded circle.

the parent of B. Consequently, B is a child of A. If there is a directed chain of nodes, the node is an ancestor to another node if it is earlier in the chain. If it is later in the chain it is called descendant. One useful concept is a Markov blanket of a node (Figure 5.). Markov blanket consists of the node's parents, its children, and its children's parents. A node without parents is called the root node, and node without children is called the leaf node.

We have defined a directed acyclic graph and how it represents random variables and their dependencies. To have a BN we quantify dependencies between variables. That relationship is quantified with a conditional probability distribution (CPD) associated with each variable. The conditional probability can be mathematically presented as

$$P(X|Y) = \frac{P(X \cap Y)}{P(Y)}. \tag{2.6}$$

That is, given event $Y$ has occurred, the probability that event X will occur is $P(X|Y)$ [16]. The conditional probability distribution, then, is telling us how the probabilities of some variable X are distributed regarding variables it depends on. After we got a structure of the network it is necessary to define the CPD, which in case of discrete variables takes the

form of a conditional probability table (CPT). For doing that we must look at all possible combinations of the values of the parent's nodes. Each such combination is called the instantiation of a parent set [16]. For every instantiation of parents, we should specify the probability that the child will take each of its values.

Modeling of Bayesian networks requires the assumption of Markov property. This assumption is that there are no direct dependencies in the system being modeled which are not already explicitly represented with arcs [16]. In other words, CPD of a node depends only on its parents [19]

We can consider BN to be a representation of joint probability distribution (JPD) of the system modeled. If we can represent a system in a compact way so that not every node is connected we have a computationally tractable representation. In a network with $n$ nodes we can represent the CPD as $P(x_1, x_2, \ldots, x_n)$. With (2.2), extended to more than two variables, we can factorize JPD as:

$$P(x_1, x_2, ..., x_n) = \prod_i P(x_i|x_1, \ldots, x_{n-1}). \tag{2.7}$$

Considering Markov property, which tell us that value of particular node is dependent only on values of its parents, we can reduce (2.7) to:

$$P(x_1, x_2, ..., x_n) = \prod_i P(x_i|Parents(x_i)). \tag{2.8}$$

## 2.3 Inference in probabilistic models

One of the main tasks of BN is the computation of posterior probabilities for query nodes, given some evidence. This task is often called probabilistic inference, or belief updating [16]. There are 2 major types of inference algorithms: exact and approximate.

The first inference algorithm was proposed by Pearl in [17]. It was an exact algorithm and it is called belief propagation algorithm today. It formed a basis for other exact and approximate algorithms. This algorithm worked only on specific types of networks called trees. Trees are network structures that permit only one undirected path between any two nodes. If any node has more than one parent the network has polytree structure. The belief propagation was followed immediately with a polytree algorithm which could do inference on polytrees also. One of the main significance of this algorithm is that it showed how important is independence in reducing the complexity of inference.

Researchers wanted to find an algorithm that will generalize inference to an arbitrary structure. One of the main ideas that came from this direction is based on conditioning. Conditioning here means setting some variables to some specific value. If you repeat the condition for every possible value for some variables you can simplify network structure.

Pearl proposed an algorithm known as loop-cutset. In this algorithm, you condition on enough variables to make network structure a polytree.

The first algorithm that found widespread use and it is used still today is a joint tree algorithm. Today it is used in many commercial implementations of BNs. The way it works is that it is a polytree algorithm on a tree of clusters. For it to work you need to convert the direct acyclic graph to the tree of clusters of nodes while meeting some requirements. The main disadvantage is that it is not feasible for networks with large treewidth.

While searching for alternatives to joint tree algorithm researches developed a variable elimination algorithm. The main advantage of this algorithm is its simplicity. This algorithm is used for inference for predicting intention in this approach after the network was trained.

Approximate algorithms are showed to be more efficient from exact algorithms although less accurate. Shortly after his first exact algorithm, Pearl proposed Gibbs sampling as an approximate algorithm for bayesian inference. This paper became a basis for other Markov Chain Monte Carlo based methods.

Another approach Pearl proposed was using a polytree algorithm for inference on networks with arbitrary structures. This algorithm is called loopy belief propagation (LBP). With some modifications, it is showed that it can be used on arbitrary networks and that it converges with good approximations. The dilemma about what is it converging to inspired lots of papers. According to first characterization, LBP is an approximate distribution of a BN that has a polytree structure. Iterations of this algorithm were finding node marginals that are minimizing KL-divergence between original and approximate distributions. KL - divergence represents disimilarity of probability distributions.

Turns out that LBP is part of a broader type of approximation algorithms called variational algorithms. The variational algorithm turns approximate inference to an optimization problem. It works by first assuming a tractable class of approximate distributions and trying to find parameters that minimize KL-divergence between approximate and original distribution.

## 2.4 Black Box Variational Inference

In this section we will show Black Box Valiational Inference (BBVI) algorithm from [20]. BBVI is a variational inference algorithm developed to solve the inference problem on different models. Variational inference tries to find parameters of simple distributions so that distribution is close to real posterior. These approximate distributions are called variational distributions. We quantify the closeness of distributions with Kullback-Leiber (KL) divergence. For an arbitrary model, practitioners have to develop specific algorithms

for variational inference. This makes modeling hard because the practitioner is checking assumptions iteratively when creating a model.

Variational methods generally work by converting posterior estimation to optimization problem [20]. While optimizing parameters are updated to adjust variational distribution to be similar to real posterior. In BBVI gradients of the objective function is the expectation of function $f$ of the latent and observed variable. BBVI optimize objective by:

1. sampling from variational distribution,
2. evaluating function $f$,
3. computing Monte Carlo estimates of the gradient
4. using these gradients in stochastic optimization to optimize parameters.

Let us define a probabilistic model (Figure 6) and some terms we need to derive BBVI. Let $x$ be observations, $z$ latent variables, $\lambda$ variational parameter and $q(z|\lambda)$ variational distribution of a probabilistic model.

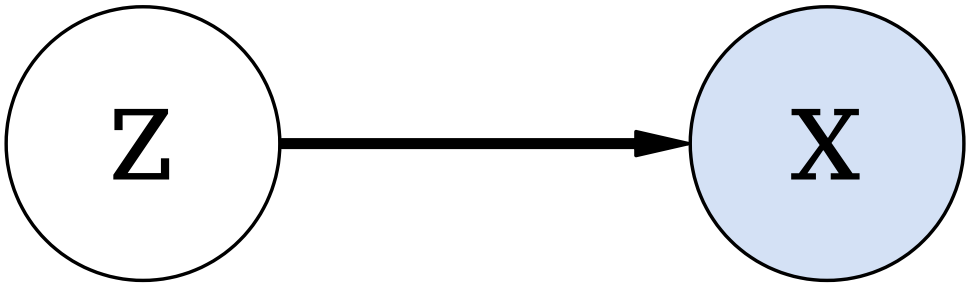


**Figure 6. Probabilistic model**

Probabilistic model used for deriving black box variational inference algorithm. The variable x represents observations and z latent variables. The standard notation when drawing probabilistic graphical models is to color observed nodes as in figure.

**Expectation** Let us first define the term expectation of a discrete random variable. We say that expectation of random variable $X$ which can take values $x_1, x_2, ...$, and have probability mass function $p$ is:

$$\mathbb{E}[X] = \sum_i x_i P(X = x_i) = \sum_i x_i p(x_i). \tag{2.9}$$

From this, we can say that expectation is a weighted average of values where weights are probabilities of those values [21].

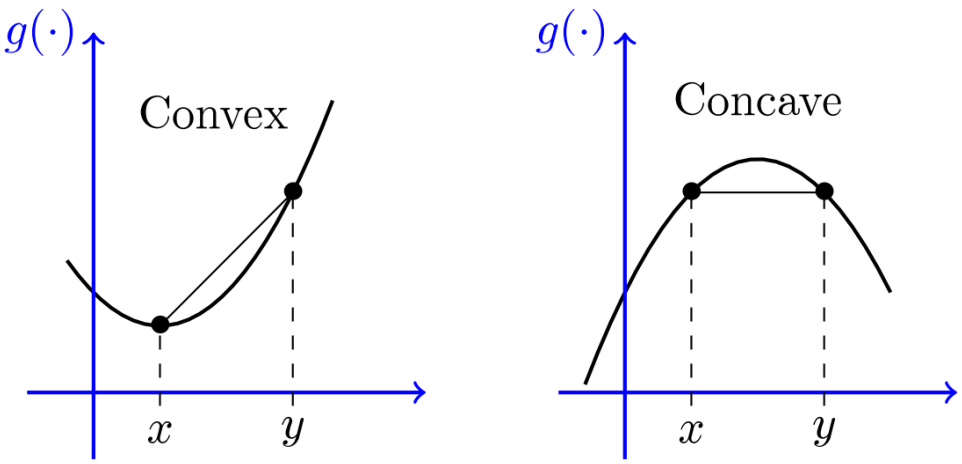


**Figure 7. Convex and Concave functions**

Extending this definition to continuous variables with probability density function $f$ gives:

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} xf(x)dx. \tag{2.10}$$

**Jensen Inequality** Jensen inequality comes from definition of convex function We say that function $g$ is convex if, for any two points $x$ and $y$ with $\alpha = [0, 1]$ :

$$g(\alpha x + (1-\alpha)y) \leq \alpha g(x) + (1-\alpha)g(y). \tag{2.11}$$

Function is concave if:

$$g(\alpha x + (1-\alpha)y) \geq \alpha g(x) + (1-\alpha)g(y). \tag{2.12}$$

We can generalize these expressions for more than two values $(x_1, x_2, ..., x_n)$. Concave function then is:

$$g(\alpha_1 x_1 + \alpha_2 x_2 + ... + \alpha_n x_n) \geq \alpha_1 g(x_1) + \alpha_2 g(x_2) + ... + \alpha_n g(x_n) \tag{2.13}$$

For variable $X$ with possible values $x_1, x_2, ..., x_n$ we can set that

$$\alpha_i = P(X = x_i) = p(x_i). \tag{2.14}$$

Combining (2.9), (2.13), (2.14), assuming that $g$ is concave, we get Jensen inequality:

$$g(\mathbb{E}[X]) \geq \mathbb{E}[g(X)]. \tag{2.15}$$

**Evidence Lower Bound** Starting from log probability of observations we have:

$$\log p(X) = \log \int_Z p(X, Z). \tag{2.16}$$

Let's expand that:

$$\log p(X) = \log \int_Z p(X, Z) \frac{q(Z)}{q(Z)}. \tag{2.17}$$

With (2.9) we can transform (2.17) to:

$$\log p(X) = \log \left( \mathbb{E}_q \left[ \frac{p(X, Z)}{q(Z)} \right] \right). \tag{2.18}$$

Since we know that log function is concave, we can use Jensen inequality (2.15):

$$\log \left( \mathbb{E}_q \left[ \frac{p(X, Z)}{q(Z)} \right] \right) \geq \mathbb{E}_q \left[ \log \frac{p(X, Z)}{q(Z)} \right] \tag{2.19}$$

Using logarithm quotient rule we can write:

$$\log p(X) \geq \mathbb{E}_q \left[ \log p(X, Z) \right] + H[Z], \tag{2.20}$$

Where

$$H[Z] = -\mathbb{E}_q[\log q(Z)] \tag{2.21}$$

is called Shanon entropy.

Right hand side of equation (2.20) is called Evidence Lower Bound or Variational lower Bound. Let us denote it as:

$$L = \mathbb{E}_q \left[ \log p(X, Z) \right] + H[Z] \tag{2.22}$$

**Kullback-Leiber divergence** We use Kullback-Leiber divergence to measure difference between distributions. KL divergence represents amount of information required to distort distribution $p(Z)$ to $q(Z)$. KL divergence:

$$KL\left[q(Z)|p(Z|X)\right] = KL = \int_Z q(Z) \log \frac{q(Z)}{p(Z|X)} = -\int_Z q(Z) \log \frac{p(Z|X)}{q(Z)}. \tag{2.23}$$

Using (2.6) we can transform right side as:

$$KL = -\left( \int_Z q(Z) \log \frac{p(X, Z)}{q(Z)} - \int_Z q(Z) \log p(X) \right). \tag{2.24}$$

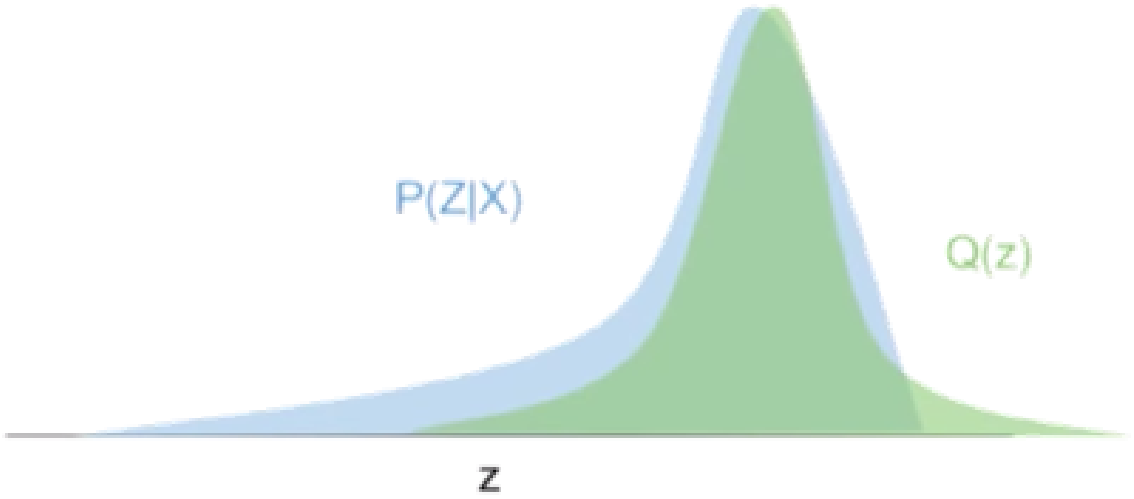


**Figure 8. Kullback-Leiber divergence. Taken from [22]**

Kullback - Lieber divergence is used to measure difference between distributions. The closer two distributions get to each other, the lower the loss becomes. In the graph above, the green distribution is trying to model the blue distribution. As the green distribution comes closer and closer to the blue one, the KL divergence loss will get closer to zero.

$$KL = -\int_Z q(Z) \log \frac{p(X,Z)}{q(Z)} + \log p(X) \int_Z q(Z) \tag{2.25}$$

Taking in consideration equation (2.22) and normalization constraint

$$\int_Z q(Z) = 1, \tag{2.26}$$

we can rearange equation (2.25) as:

$$KL\left[q(Z)|p(Z|X)\right] = -L + \log p(X). \tag{2.27}$$

From equation (2.27) it is obvious that minimizing KL - divergence is the same as maximizing ELBO.

The goal of BBVI is to maximize ELBO (minimize KL - divergence) to make variational distribution closest to real posterior. To maximize ELBO this approach uses stochastic optimization. Stochastic optimization uses noisy estimates of gradients to update parameters. Stochastic optimization updates $x$ in interation $t$ as:

$$x_{t+1} \leftarrow x_t + \rho_t h_t(x_t). \tag{2.28}$$

Where $f(x)$ is objective function, $h_t$ is specific value of $H(x)$. $H(x)$ is random variable with expectation $f(x)$. $\rho$ is learning rate. This procedure converges if the following rules are met:

$$\sum_{t=1}^{\infty} \rho_t = \infty, \tag{2.29}$$

$$\sum_{t=1}^{\infty} \rho_t^2 < \infty. \tag{2.30}$$

These rules are known as Robbins-Monro conditions.

To optimize ELBO with stochastic optimization we need to compute unbiased gradents from samples of variational distribution. Differentiating (2.22) we get:

$$\nabla_\lambda L = \nabla_\lambda \int (\log p(x,z) - \log q(z|\lambda)) q(z|\lambda) dz \tag{2.31}$$

Using dominated convergence theorem [23] we can write:

$$\nabla_\lambda L = \int \nabla_\lambda [(\log p(x,z) - \log q(z|\lambda)) q(z|\lambda)] dz. \tag{2.32}$$

Using product rule this becomes

$$\nabla_\lambda L = \int \nabla_\lambda [(\log p(x,z) - \log q(z|\lambda))] q(z|\lambda) dz \tag{2.33}$$

$$+ \int \nabla_\lambda q(z|\lambda) (\log p(x,z) - \log q(z|\lambda)) dz \tag{2.34}$$

With $\nabla_\lambda [q(x,z)] = 0$ and (2.10)

$$\nabla_\lambda L = -\mathbb{E}_q [\nabla_\lambda \log q(z|\lambda)] + \int \nabla_\lambda q(z|\lambda) (\log p(x,z) - \log q(z|\lambda)) dz \tag{2.35}$$

It can be showed that first term in (2.35) is equal to zero

$$\mathbb{E}_q [\nabla_\lambda \log q(z|\lambda)] = \mathbb{E}_q \left[ \frac{\nabla_\lambda q(z|\lambda)}{q(z|\lambda)} \right] \tag{2.36}$$

$$= \int \nabla_\lambda q(z|\lambda) dz \tag{2.37}$$

$$= \nabla_\lambda \int q(z|\lambda) dz = \nabla_\lambda 1 = 0 \tag{2.38}$$

If we observe that:

$$\nabla_\lambda [q(z|\lambda)] = \nabla_\lambda [\log q(z|\lambda)] q(z|\lambda), \tag{2.39}$$

we get gradient of ELBO as expectation

$$\nabla_\lambda L = \mathbb{E}_q \left[ \nabla_\lambda \log q(z|\lambda)(\log p(x,z) - \log q(z|\lambda)) \right]. \tag{2.40}$$

With (2.40) we can calculate noisy unbiased gradints with Motne Carlo approximation

$$\nabla_\lambda L = \frac{1}{S} \sum_{s=1}^{S} \nabla_\lambda \log q(z_s|\lambda)(\log p(x,z_s) - \log q(z_s|\lambda)) \tag{2.41}$$

where $z_s \sim q(z|\lambda)$.

Only assumption for using this algorithm is that practicioner can compute log of $p(x, z_s)$. This way it is easy to implement variational inference to broad range of models.

## 2.5 Probabilistic Programing

Bayesian machine learning uses probability calculus as the main tool. In contrast, Deep learning practitioners don't regard probability calculus as the main tool for building models. The author of [5] argues that one of the reasons for the rapid expansion of deep learning methods is automatic differentiation. He also argues that without a similar toolchain for Bayesian machine learning, that rapid expansion is not probable. Probabilistic programming aims to deliver such a toolchain.

You can look at probabilistic programming as a way to automate Bayesian inference. Probabilistic programming combines fields of statistics, machine learning, and programming languages. It uses inference algorithms from statistics and semantics, compilers and other tools from programing languages [5]. The goal of a programming language is to build inference evaluators for models and applications in machine learning.

Another way to describe probabilistic programming is by doing statistics with computer science tools. The standard pipeline of a computer science program can be stated as:

1. Program gets some input,
2. Program evaluates input,
3. Program returns output.

In statistics, this pipeline is inversed. Here you have output, which is some observations or data $y$. You specify generative model $P(x, y)$ which can generate statistically similar data. You use the appropriate inference algorithm to figure out posterior distribution $P(x|y)$

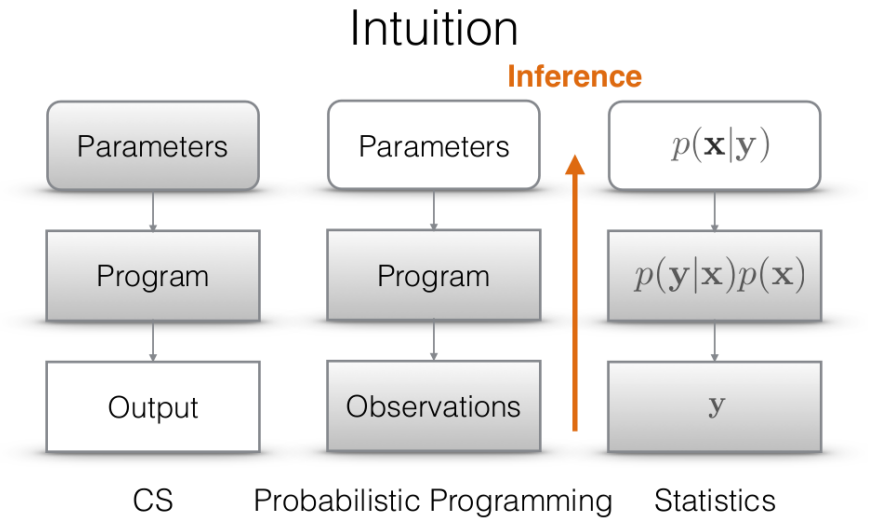


**Figure 9. Comparison of standard and probabilistic programming pipeline [5]**

This diagram shows intuitive difference between probabilistic and regular program. Standard program returns some output according to program parameters. Probabilistic program, however, tries to find parameters according to output it could produce.

Probabilistic programming is used for performing Bayesian inference with computer science tools. The benefits of programming languages make denoting a model easier. You use inference algorithms to figure out the conditional distribution. The conditional distribution which you are looking for is one that gives observed outputs given some input to a probabilistic program.

There is a difference between programming languages and probabilistic programming languages. The probabilistic programming language must have the ability to draw values at random. However, not every language that can draw random values is a probabilistic programming language. The main thing that differentiates probabilistic programming language and programming language is conditioning. Conditioning means getting posterior distribution by encoding observations in a probabilistic program.

To see the power of probabilistic programming we will look at the example of breaking captchas. It is showed that this problem can be denoted in probabilistic programming language and solved with inference. Let us compare probabilistic and nonprobabilistic approaches applied to this problem. The nonprobabilistic approach requires a large number of captcha images with labels. This data would be fed into a Neural network which would learn to map images to strings. The probabilistic approach would require building a model that generates statistically similar images from strings. This is called the generative model. When the model is created you would condition on observed images and get posterior distribution over strings [5].

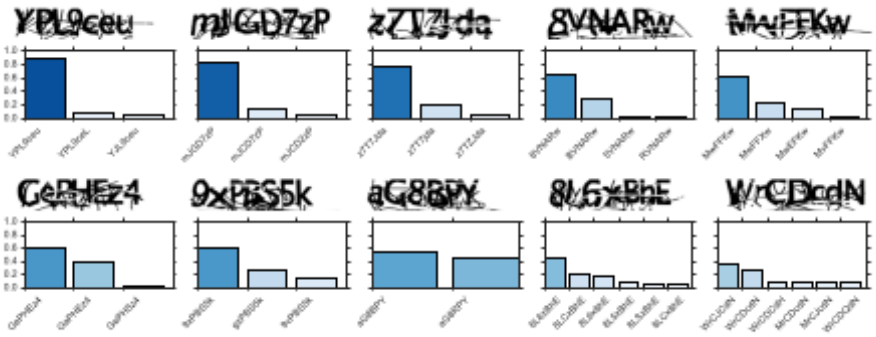

**Figure 10. Captcha breaking with probabilistic programing [5]**

Writing good models is generally a hard task. If we have a large amount of data, however, we can use this data to learn a generative model. Recently a lot of data-driven approach for modeling is outperforming expert-driven systems. A Bayesian network can be a compromise between these two approaches. The topology of the network can be designed by an expert, and then the network can learn parameter values from data. We say that we use a top-down approach when a model is created from latent variables towards observations. In contrast bottom down approach is used in deep learning, where we start from observations to compute parameters of a network.

### 2.5.1 Pyro probabilistic programing language

Pyro is a probabilistic programming language built on python [24]. It is used for developing advanced probabilistic models. Pyro leverages SVI algorithms and probability distributions built on top of PyTorch to scale to large datasets and high dimensional models. Pyro programs are written as python functions and it has only two extra language primitives. Those are `pyro.sample` and `pyro.param`.

Probabilistic graphical models are created from joining probability distributions. Probabilistic programming languages have probability distributions as stochastic functions. From these functions, it is easy to draw samples. We used Normal and Bernoulli distributions for creating nodes in our network.

A normal distribution is used for continuous variables. It is a simple distribution with two parameters:

$$\mathcal{N} \sim (\mu, \sigma) \tag{2.42}$$

where $\mu$ is a mean and $\sigma$ is standard deviation of distribution $\mathcal{N}$.

Bernoulli distribution is used for discrete binary outcomes. The output of a Bernoulli distribution is 0 or 1. It takes one parameter ($\mu$) which defines the probability that the

```
x = pyro.sample("sample_name",
    pyro.distributions.Normal(0,1)).
```

**Figure 11. Pyro sample statement**

Pyro sample statement is a builiding block of probabilistic models. This method takes two arguments. First one is the name of the sample site, and other is distribution from which samples are made. Every sample site represents random variable. This means that sample sites are nodes in our network.

```
def model():
    x = pyro.sample("x", dist.Normal(0,1))
    y = pyro.sample("y", dist.Normal(x,0.1))
    return y
```

**Figure 12. Defining model in pyro**

Random variables are connected in a model. This model is represented in pyro as a function with sample sites. Since pyro is built on top of python programming language we can use control flow, recursion, and modularity of python for modeling.

outcome is 1.

$$\mathcal{B} \sim (\mu). \tag{2.43}$$

The main building block of models in pyro is `pyro.sample` primitive. In addition to returning a sample from distribution pyro.sample gets a unique name (Figure 11.
Pyro backend uses names of sample sites for implementing manipulations on them.

Probabilistic models are combination of stochastic and deterministic computations. In pyro we create model as a function. Simple example of a pyro model is shown on figure 12:

The real power of pyro lies in its possibility of conditioning on observed data. We condition a generative model on observed data to infer latent variables that produced observations. In pyro, there are few ways to do conditioning. We can add `obs` argument to `pyro.sample` method and pass values of observed data to that argument. There is also function `condition` which takes model and data as arguments and it condition model to passed data.

Parametrized variational distribution $q(z|x)$ is called `guide()` in pyro. The parameters $\lambda$ we represent with `pyro.param()` primitive. We use it to represent trainable parameters which will be updated in optimization. Every `pyro.param()` is named so that pyro backend can manipulate parameters at runtime. Figure 14 shows example of a `guide()` with trainable parameter.

There are few constraints when creating a guide. The guide and the model have to had same arguments. For every unobserved sample in model we need to create same named

```
def model():
    x = pyro.sample("x", dist.Normal(0,1))
    y = pyro.sample("y", dist.Normal(x,0.1),
       obs=torch.tensor(0.2))
```

**Figure 13. Conditioned model in pyro**

Conditioning is the main thing that differentiates probabilistic programming languages from standard programming languages. In pyro conditioning is done with `obs` keyword passed to sample sites we want to observe.

```
def guide():
    mu = pyro.param("mu_param", torch.tensor(0.))
    x = pyro.sample("x", dist.Normal(mu,1))
    y = pyro.sample("y", dist.Normal(x,0.1))
```

**Figure 14. Defining guide (variational distribution) in pyro.**

Guide represents variational distribution from SVI. Trainable parameters are presented as pyro.param method.

sample in the guide. Also, guide must not have any observed values. This makes sampling in guide straightforward. To sample guide you only need to forward run the program.

### 2.5.2 SVI in pyro

Black box variational inference is implemented in pyro. The paper [25] presents Automated Variational Inference which pyro used to implement BBVI. In section 2.4 we derived BBVI and here we will show how is it implemented in pyro.

Probabilistic model defined as $p(x, z)$ is a stochastic function `model()`. The goal is to find marginals of latent values given some observations $p(z|x)$. We can condition `model()` with adding `obs` argument to `sample()` variables that are in the `model()`. The variational distribution $q(z|\lambda)$ is also a stochastic function called `guide()` in pyro. One creates `guide()` from `model()`.

One easy way to create variational distribution is a partial mean-field approximation. Here this means that the `model()` is run forward and every time `sample()` is encountered, variational parameter is used for sampling instead of original parameter [25]. This means that ELBO is constructed with respect to variational distribution as in (2.22). In pyro first the `guide()` trace is populated. When we have all samples from `guide()` we run `model()`. Every time a unobserved sample is encountered in `model()`, we replace it with corresponding sample from a `guide()`.

This can be extended for automated calculation of stochastic gradients. The procedure looks as following:

1. Get guide trace.
2. Get model trace with samples replaced as described above.
3. Compute log prob of a `guide()`.
4. Compute log prob of a `model()`.
5. Compute ELBO.
6. Compute Gradients.

# 3 Data

The dataset used for training a model in this thesis is the highD dataset [26]. HighD is a dataset of vehicle trajectories recorded in Germany in 2018. Recordings are made by a drone with a camera. This approach overcomes some limitations of typical ground-based data collection methods. For example, since the images are captured from aerial perspective there are no occlusions and every vehicle is visible.

The kinematical features are extracted automatically using computer vision. The U-Net neural network is used for segmentation and from that bounding boxes are created. Static objects are manually labeled. The authors report position error of 10 cm.

This dataset includes 16.5 hours of recordings from six different locations. The data is formatted in the following way. Every recording is described with four files. The first file is a picture of the location recording is made. Second file is a `.csv` file which includes location, traffic signs, speed limit, driving lanes, and similar static data about recording. The third file contains a summary of every vehicle. This means features like maximum and minimum velocity, height, width, etc. The last file is about kinematic data about vehicles in every frame and its relation to other participants.

## 3.1 Raw data exploration

Let us explore our dataset. As an example we will look at recording number 23 (Figure 15).

**Figure 15. Shot of the highway for recording 23.**

Highway as recorded with a drone and with vehicles removed in post processing. Every direction has two lanes. Recorded part of a highway is around 400 m long.

The 23_tracks.csv file contains data collected 25 times per second. In table 1 we can see what fields does dataset contains with descriptions and units.

In table 2 the summary statistics can be seen for kinematics of vehicles. To better understand these values, let us visualize distributions of these values. In Figure 16 histograms of velocities and accelerations of vehicles are presented. From the distribution of longitudinal velocity (xVelocity) asymmetry of the values, caused by two directions of is best seen. We can not conclude the same from other distributions without dividing plots to separate lanes for example.

All distributions are similar to normal distribution around mean of zero except the distribution of longitudinal velocity. If we compare visualization of longitudinal velocity

**Table 1. Description of fields in tracks.csv files.**

| | | |
|---|---|---|
| frame | current frame. | [-] |
| id | track's id. | [-] |
| x | x position | [m] |
| y | y position | [m] |
| width | width of the vehicle. | [m] |
| height | height of the vehicle. | [m] |
| xVelocity | longitudinal velocity. | [m/s] |
| yVelocity | lateral velocity. | [m/s] |
| xAcceleration | longitudinal acceleration. | [m/s] |
| yAcceleration | lateral acceleration | [m/s] |
| frontSightDistance | distance to the end of the recorded highway. | [m] |
| backSightDistance | distance to the begining of the recorded highway. | [m] |
| dhw | Distance Headway. | [m] |
| thw | Time Headway. | [s] |
| ttc | Time-to-Collision. | [s] |
| precedingXVelocity | longitudinal velocity of the preceding. | [-] |
| precedingId | id of the preceding vehicle in the same lane. | [-] |
| followingId | id of the following vehicle in the same lane. | [-] |
| leftPrecedingId | id of the preceding vehicle on the adjacent lane on the left. | [-] |
| leftAlongsideId | id of the adjacent vehicle on the adjacent lane on the left. | [-] |
| leftFollowingId | id of the following vehicle on the adjacent lane on the left. | [-] |
| rightPrecedingId | id of the preceding vehicle on the adjacent lane on the right. | [-] |
| rightAlsongsideId | id of the adjacent vehicle on the adjacent lane on the right. | [-] |
| rightFollowingId | id of the following vehicle on the adjacent lane on the right. . | [-] |
| laneId | id of lanes. | |

with it summary statistics we can conclude that mean extracted from table 2 is deceiving. We see that average velocity is about 30 m/s in both directions and not -2 m/s.

It can be useful to visualize correlation of variables as seen in figure 18.

**Table 2. Summary statistics of numeric values of the dataset.**

| | x | y | xVelocity | yVelocity | xAcceleration | yAcceleration |
|---|---|---|---|---|---|---|
| mean | 199,41 | 16,38 | -2,66 | 0,01 | 0,16 | 0,00 |
| std | 118,50 | 6,45 | 28,67 | 0,16 | 0,31 | 0,07 |
| min | -20,19 | 7,33 | -46,66 | -1,36 | -1,94 | -0,62 |
| 25% | 96,66 | 12,13 | -27,26 | -0,04 | 0,02 | -0,03 |
| 50% | 198,09 | 13,09 | -22,72 | 0,01 | 0,14 | 0,00 |
| 75% | 301,37 | 21,93 | 28,96 | 0,07 | 0,30 | 0,04 |
| max | 416,01 | 27,11 | 48,90 | 1,54 | 3,57 | 0,60 |

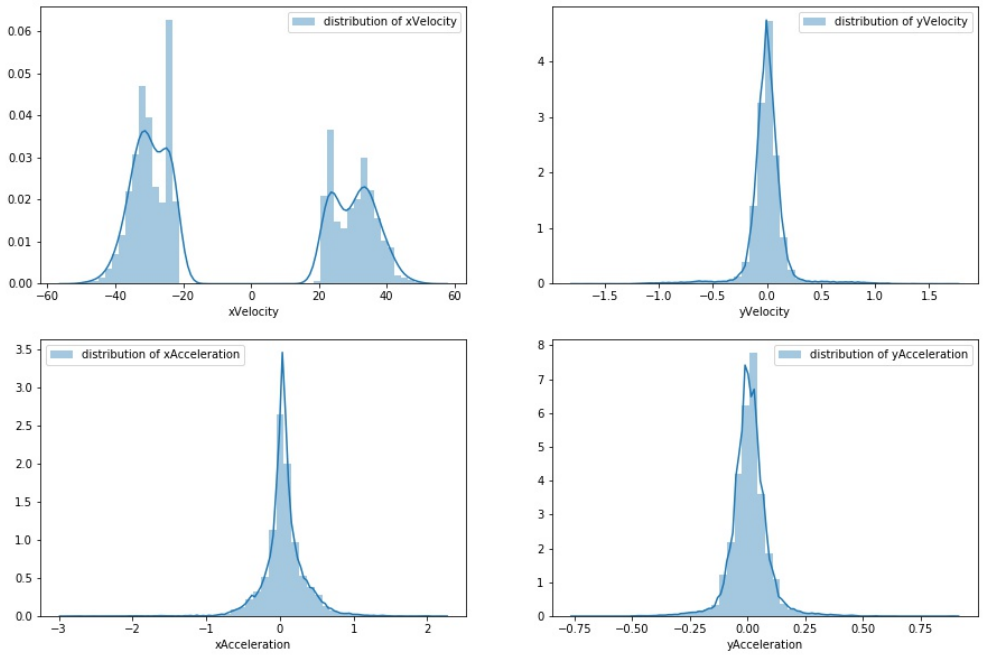


**Figure 16. Distribution of acceleration and velocity of vehicles**

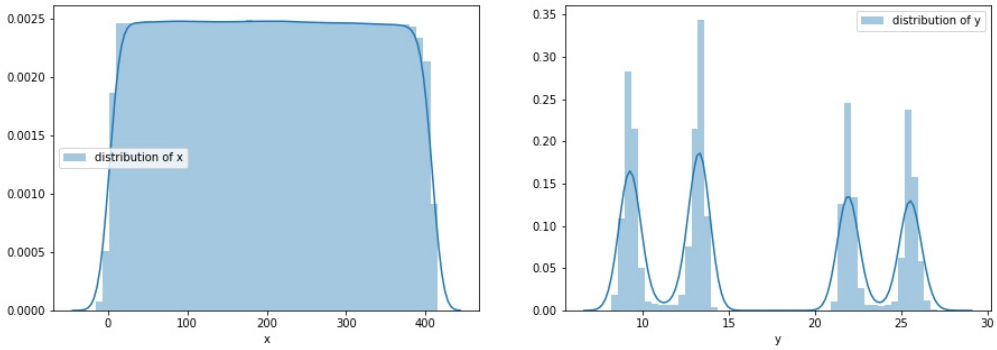


**Figure 17. Distribution of collected positition data**

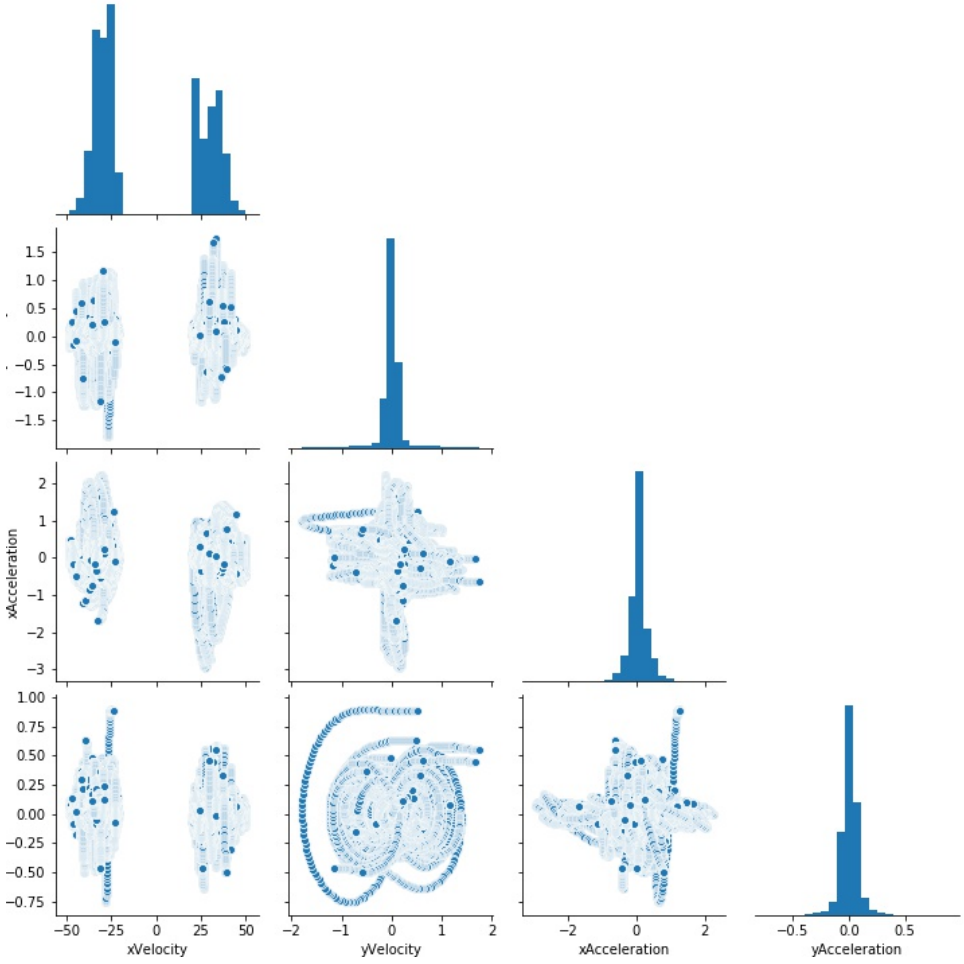


**Figure 18. Pair plots of accelerations and velocities of vehicles**

We can extract one vehicle from the dataset and visualize its trajectory and kinematic values (Figure 20). The origin of the coordinate system in which values are measured is defined to be in the top left corner of the recording. The positive longitudinal axis (x-axis) is in right direction and positive lateral direction (y-axis) is in bottom direction of a camera (Figure 19). In the same way, trajectory is visualized on figure 20.

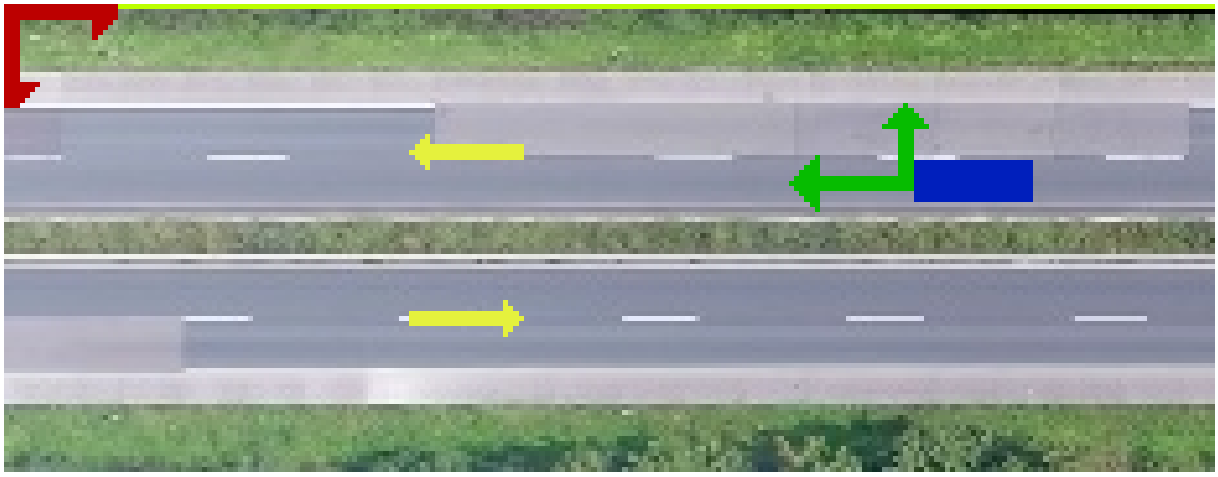

**Figure 19. Highway with coordinate systems and directions.**

The yellow lines shows in what direction vehicles are driving in every lane, global coordinate system is red, local coordinate system is green and is connected to blue vehicle.

The lanes are labeled from top to bottom with numbers: 2,3,5,6. In lanes 2 and 3 vehicles forward driving is from right to left on the recording. Lane 2 is the right lane (slow lane) here and lane 3 is the left lane (fast lane). For lanes 5 and 6 it is the other way around. Vehicles here are driving from left to right. Here lane 6 is a right lane and lane 5 is left lane. These two directions are causing asymmetry in the dataset which we will deal later in this thesis when preparing data.

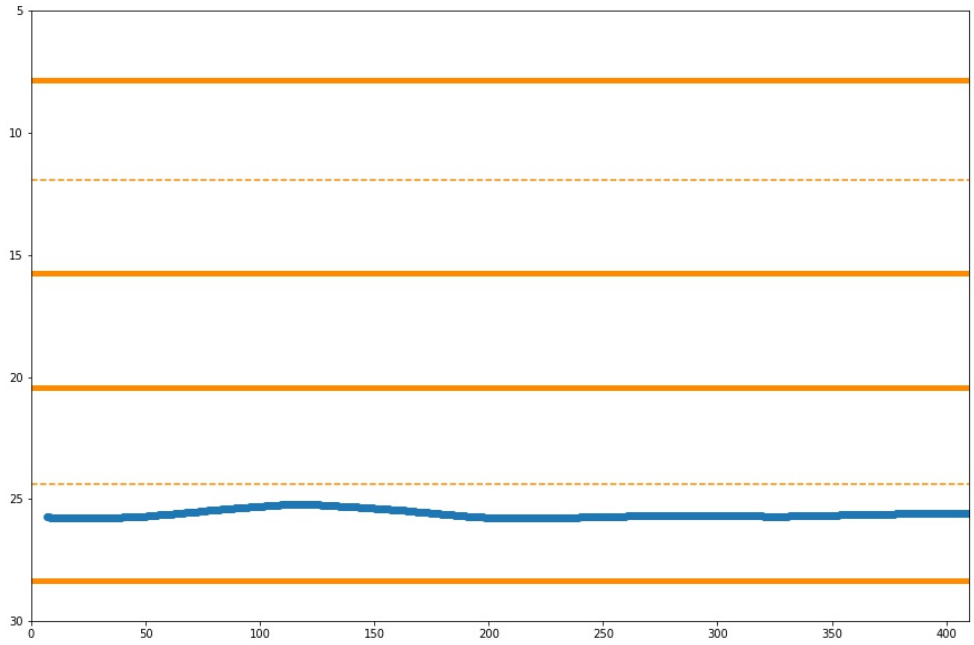


**Figure 20. Example trajectory from the dataset**

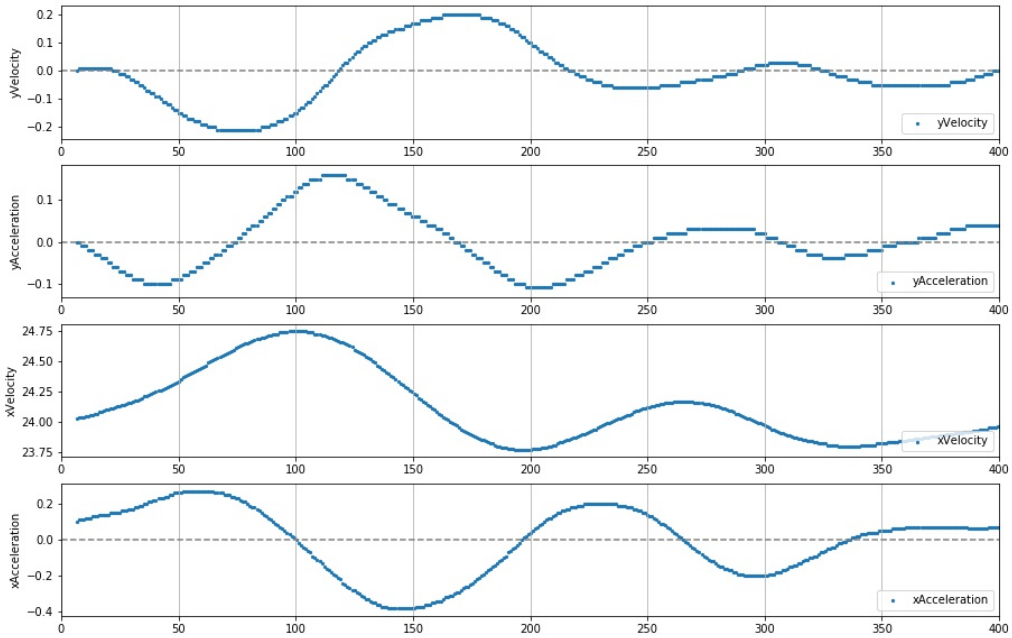


**Figure 21. Kinematics of example trajectory**

## 3.2 Data preparation

To train our model on this data we must preprocess it. Since y datapoint is coordinate of the upper left corner of the vehicle bounding box, we changed it to be coordinate of the center of a vehicle. This was achieved by adding half the height value of the vehicle to its y coordinate. Another file contains vehicle class information. This means is the vehicle a car or a truck. We added this information to our dataset for every vehicle.

Every recording contains data from two driving directions. This causes asymmetry in the dataset. To deal with this we converted all kinematic values to local coordinate systems attached to ego vehicle. After this transformation following statements are true:

- yVelocity is negative in vehicles left direction and positive in vehicles right direction.
- yAcceleration is negative in vehicles left direction and positive in vehicles right direction.
- xVelocity is positive for forward driving and (hypoteticaly) negative for reverse driving.
- yAcceleration is positive for accelerating and negative decelerating.

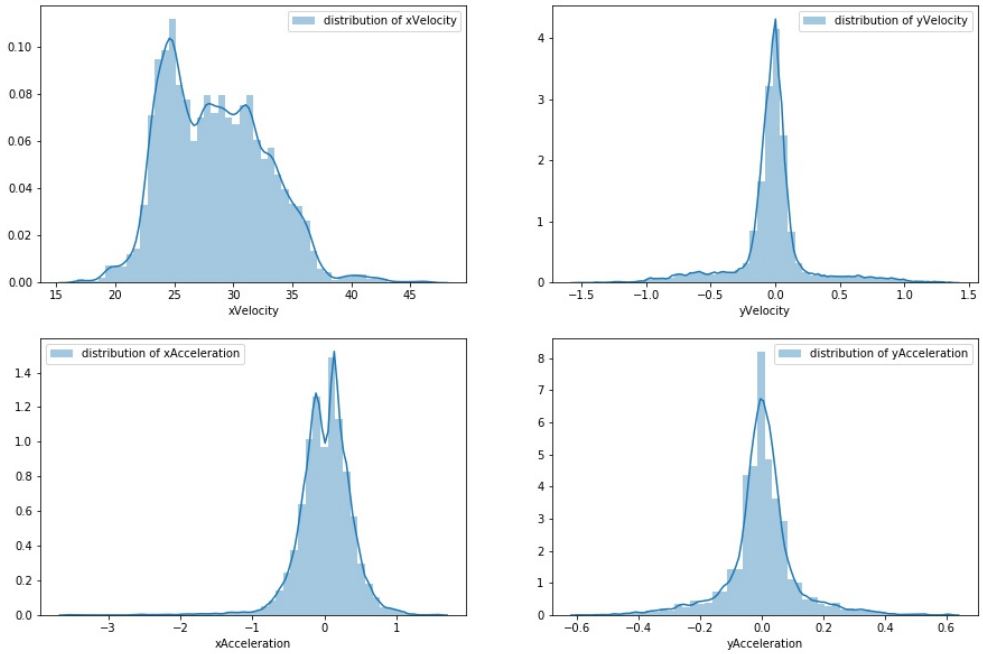


**Figure 22. Distribution of velocities and accelerations after transformation.**

The "laneId" column contains information about what lane is vehicle occupying in every frame. From this information, we can conclude if the vehicle changed lanes during recording. We created the column "laneChange" with labels about whether a vehicle changed the lane at least once. The label 1 represents at least one lane change and label 0 staying in the lane the whole time. This label is on the vehicle id level. This means that it is true for the whole trajectory if the vehicle did perform a lane change.

For predicting lane change intention we need to label actual lane changes and not the whole trajectory. Since the dataset is big this is labeled automatically. The assumption is that driver has the intention to change lane $S$ seconds before the vehicle actually touches the other lane. We created columns with $S = [1, 2, 3, ..., 8]$ seconds. The procedure is as follows:

1. Find when the vehicle "laneId" label changed.
2. Label this as lane change.
3. Propagate the label for $T$ steps in past or until begining of trajectory.
4. Propagate the label for $T$ steps in the future or until the end of trajectory.

$T$ here represents the number of frames you want to propagate labels in the past. data is collected in the resolution of 25 frames per second for every recording, the number T is $T = 25 \cdot S$.

We also created the column "laneIdRL" which contains a label about lane relations. In other words, is a vehicle in the right or left lane? Having already "laneId" labels and knowing which lane numbers are left and which are right lanes this was a straightforward task.

There are a lot more vehicles that do not perform a lane change in a dataset (Figure 23). To avoid this we reduced the number of vehicles that did not perform lane change to match the number of vehicles that performed at least one lane change. We picked random vehicles to keep.

The final step in the preparation of training data was to join all data files in one. For creating a training dataset we choose all recordings of a highway with two lanes in one direction. The following are numbers of such recordins: 1, 2, 3, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24. We joined all except number 24 in a training dataset and left recording 24 as a test data. The final training dataset had 931 753 data points.

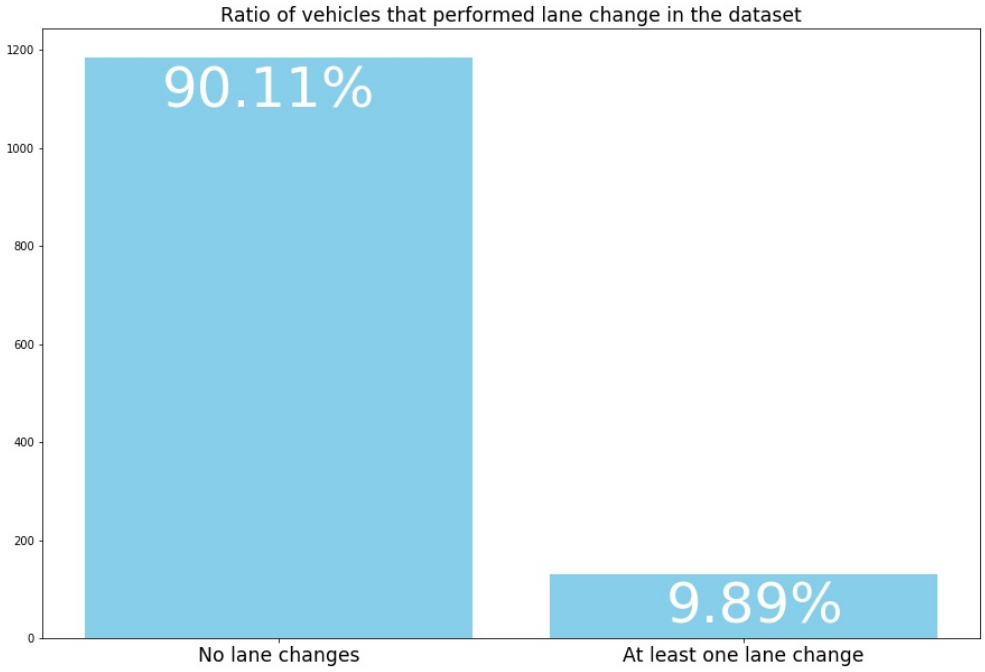


**Figure 23. Unbalanced ratio of lane change labels.**

## 3.3 Data class breakdowns

It is viable to assume that drivers behave differently in different situations. Here we will explore how driving data breaks if we divide according to some categories. The one thing that can divide behaviors on the road is the fact that you are driving a personal car or a truck. Another is about is the driver in a fast or slow lane. In addition to these two assumptions, we will explore also behavior immediately before a performed lane change.

### 3.3.1 Vehicle class

Train dataset contains $84,54\%$ of cars and $15,45\%$ of trucks. Let us see how are kinematic values distributed when divided into these two categories (Figure 24). We can see that the bimodality of xVelocity from figure 22 can be explained by different mean velocities of cars and trucks. Looking at lateral components of data we see that there is relatively less variability in trucks. The reason for this could be a tendency of truck drivers to stay in the same lane. Figure 25 shows how the smaller percent of trucks in the training dataset performs lane change even though the dataset was balanced.

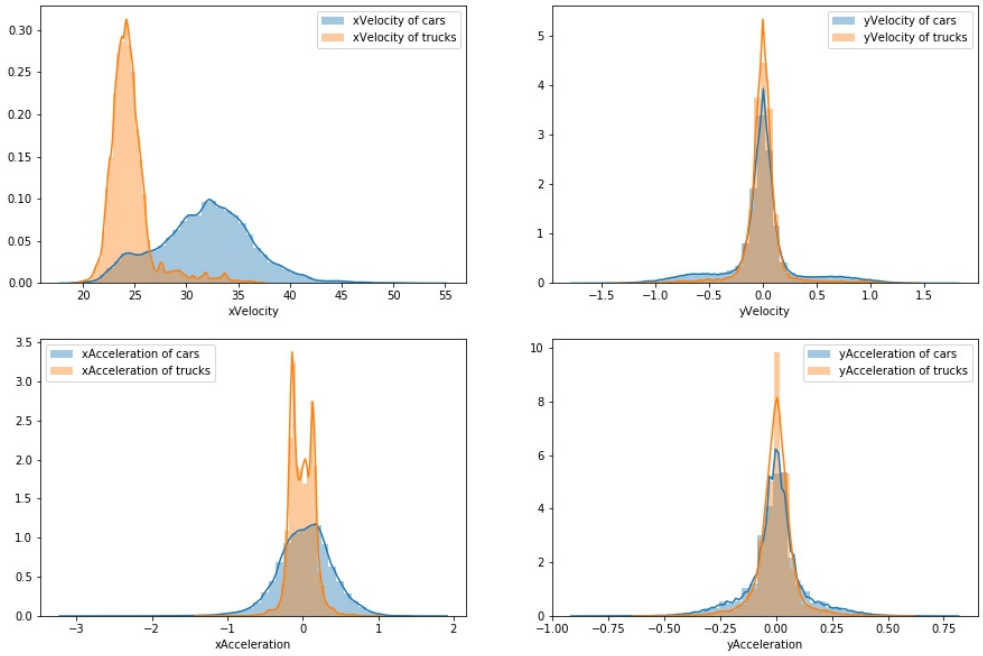


**Figure 24. Comparison of car and truck data.**

Let us also explore how type of vehicle affects frequency of lane changes. Figure 25 shows that ratio of trucks that performed lane change in a balanced dataset. This ratio is

smaller in a original dataset. The ratio of cars that performed lane change is set to 50% as described earlier.

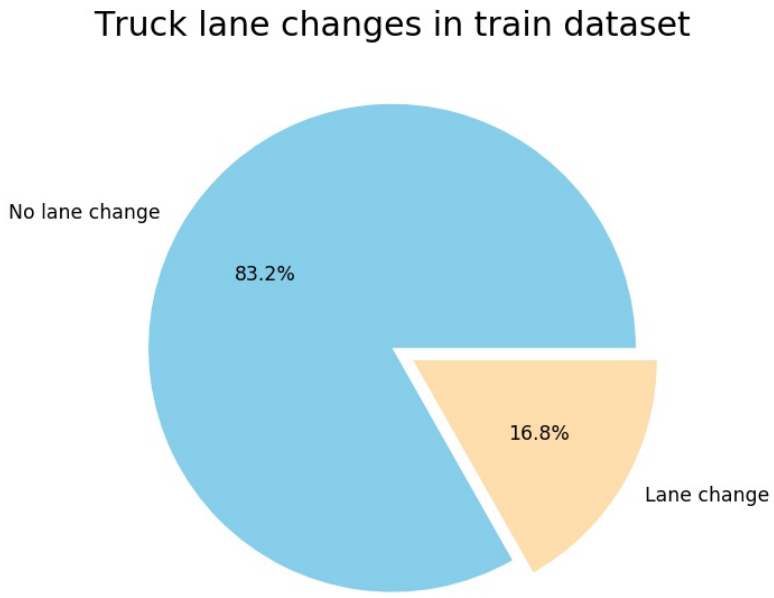


**Figure 25. Ratio of truck drivers that changed lane.**

### 3.3.2 Left and right lane

Drivers behave differently in the left and the right lane (Figure 26). We can conclude a few things by looking at longitudinal velocity distribution. Slower drivers tend to go in the right lane. Trucks spend most of their time in the right lane. We can also see from the longitudinal acceleration plot that drivers accelerate slightly more in the left lane. This is not surprising given that the left lane is used for overtaking slower vehicles.

Looking at y directed values we can see that the velocity in the y-direction is distributed almost the same in both lanes. This could be because vehicles leave and get in the lane with similar velocities. The biggest difference is in acceleration distributions. We can see that left lane distribution is shifted to positive values which represent right turning. Similarly, right lane acceleration distribution is shifted to negative values which represent left turning.

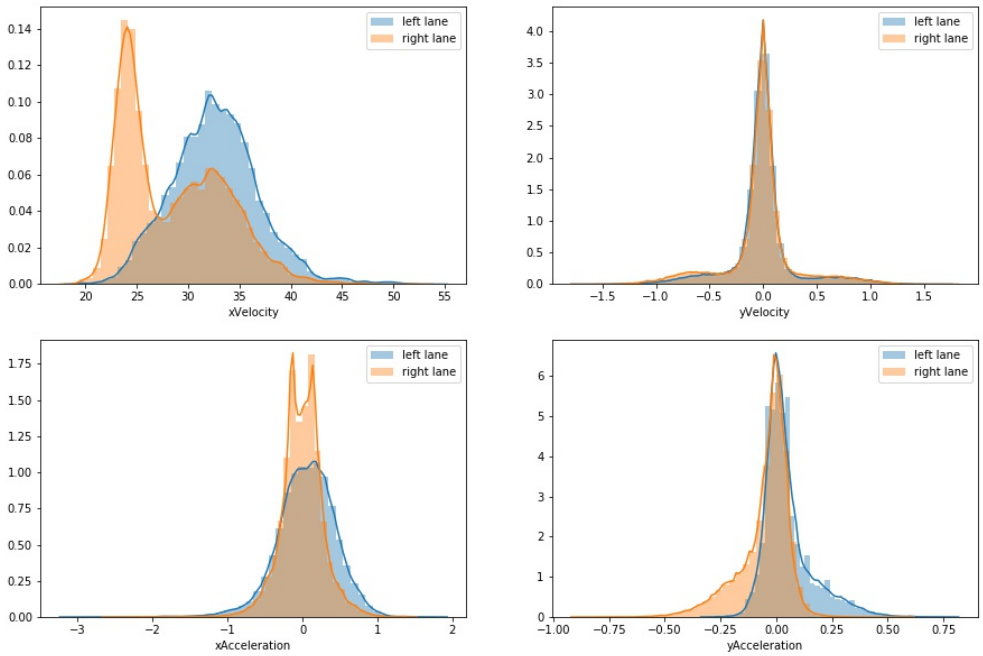


**Figure 26. Data distributions with respect to a lane.**

### 3.3.3 Lane change

The dataset comes with lane labels for every frame. This label tells us in which lane the vehicle is in a specific frame. We created a lane change label by looking if the vehicle is in the same lane for the whole trajectory. We can look now at differences between vehicles that stayed in the lane the whole time and vehicles that performed at least one lane change (Figure 27).

The difference in longitudinal velocity distribution is mostly due to the fact that trucks rarely perform lane change. We can not see major differences in the distribution of longitudinal acceleration while breaking only on lane change class. Lateral kinematics distributions are very different. The velocity and acceleration of vehicles that stayed in the same lane are narrowly distributed around zero. For vehicles that changed lanes at least one time, the values are more distributed. Almost all values of velocities are between -1,5 to 1,5 m/s. Same goes for accelerations for interval from -0,75 and 0,75 m/s$^2$.

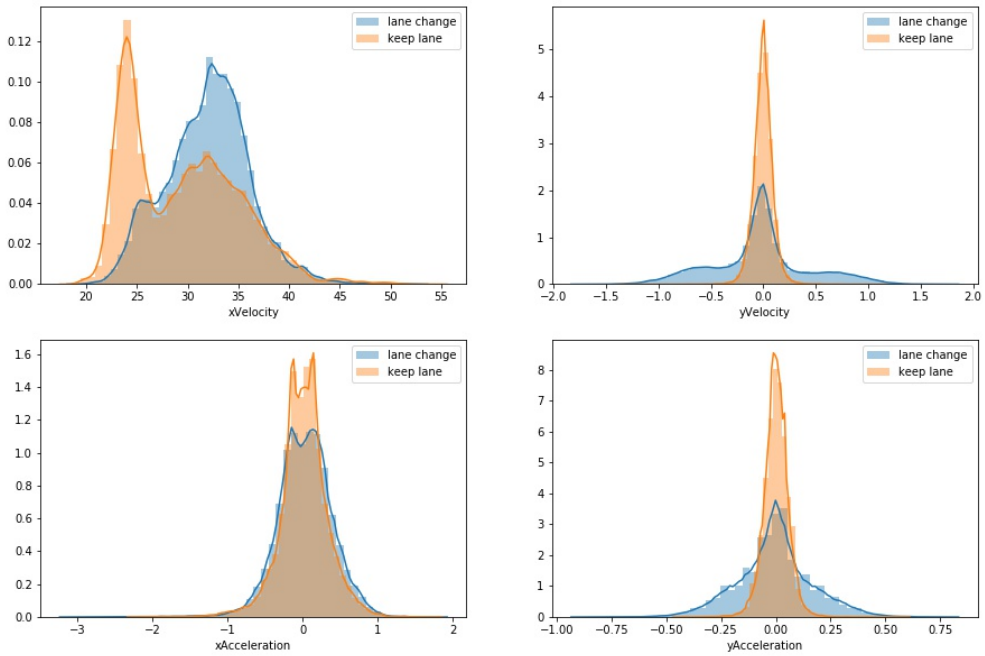


**Figure 27. Distribution of data for vehicles that changed lane at least one and vehicles that stayed in the lane for the duration of the whole trajectory**

While creating the training dataset, we labeled time intervals before vehicle changed lanes for different time lengths. For training a model we used time length of four seconds before the vehicle touches the other lane as an intention label. The column laneChangeT100 contains these labels. The four seconds correspond to 100 frames in a

dataset.

Let us see how distributions difference changes with labeling four seconds before vehicle touches another lane as lane change intention (28). We see that values for lateral kinematics got more diffuse. This means that in this time region there is more lateral movement of vehicles.

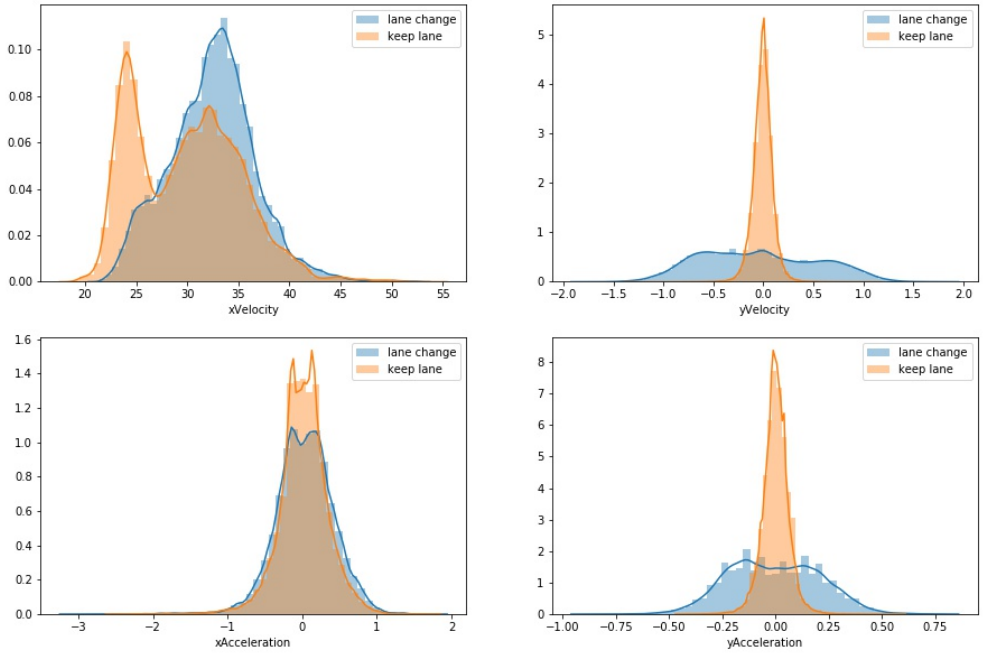


**Figure 28. Distribution of data colected four seconds before lane is changed**

To better understand differences in lateral dynamics during keeping and changing lanes we will take a closer look at the lateral data. Figure 29 shows us a scatter plot of lateral velocity and acceleration. We can see different regions forming. We can infer behaviors from this plot. When a driver is performing a lane change the vehicle's velocity is in the direction of the new lane and its acceleration is in the same direction. This is labeled with an orange color on Figure 29. The values centered about zero for both axes represent forward driving. The remaining regions of the plot are velocity in one direction with acceleration in the opposite direction. This represents vehicles that are just coming in the new lane and are trying to straighten the trajectory.

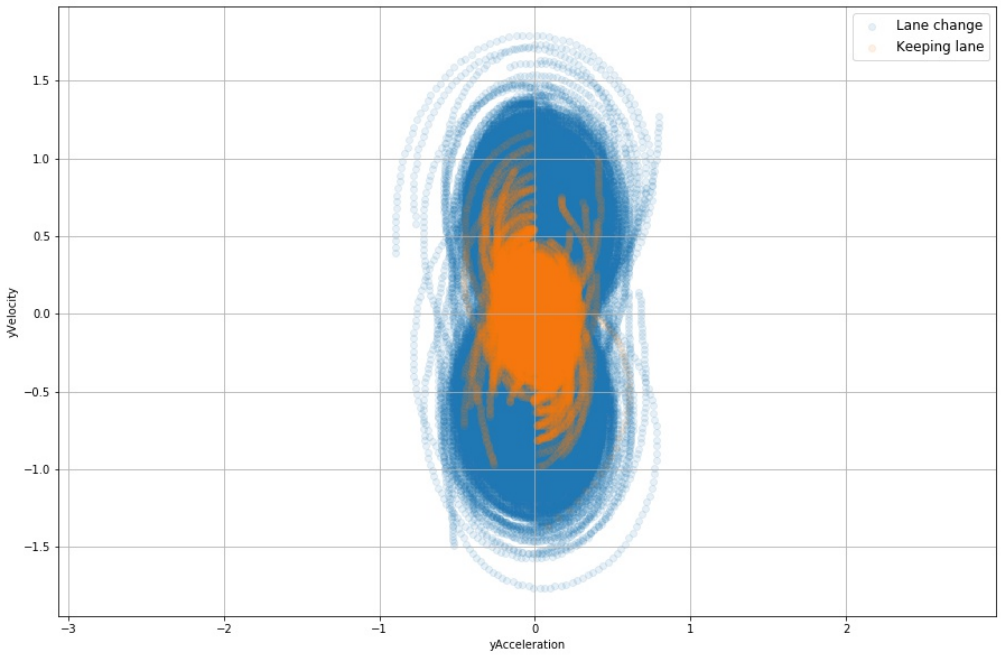


**Figure 29. Comparison of lateral kinematis for keeping and changing the lane.**

# 4 Model

In this thesis, the goal is to predict lane change intention from driver behavior using the Bayesian network. To design the Bayesian network we will use conclusions from the data exploration step. We will assume that some features affect lane change intention and that intention affects a driver's behavior. For example, we will assume that intention affects the kinematics of the vehicle. Process of creating a predictive model will be divided in two parts:

- Learning generative model.
- Infering latent intention from conditioned model.

Every probabilistic programming problem is solved by creating a generative model and then computing some values while observing others. A generative model can be designed by experts or learned from data. In this thesis, we will create the topology of the model with the knowledge extracted from data exploration . The model will learn parameters by looking at the training data set created in the data processing step. After training, we will compare prior and posterior joint probability distribution to see what the model learned.

## 4.1 Learning generative model

### 4.1.1 Network structure

Let us specify the network structure which is used for learning. Figure 31 presents the structure of our network with plate notation. Plate notation is used when we have repeating variables in a graphical model.

To understand why are variables repeating and what is plate notation we will look at an example of learning the bias of the coin. In this problem, we have data collected by noting the results of flipping a coin m times. The goal is to learn parameter $\theta$ which is a possible bias of the coin. The structure of the network used for learning will look like Figure **??**.

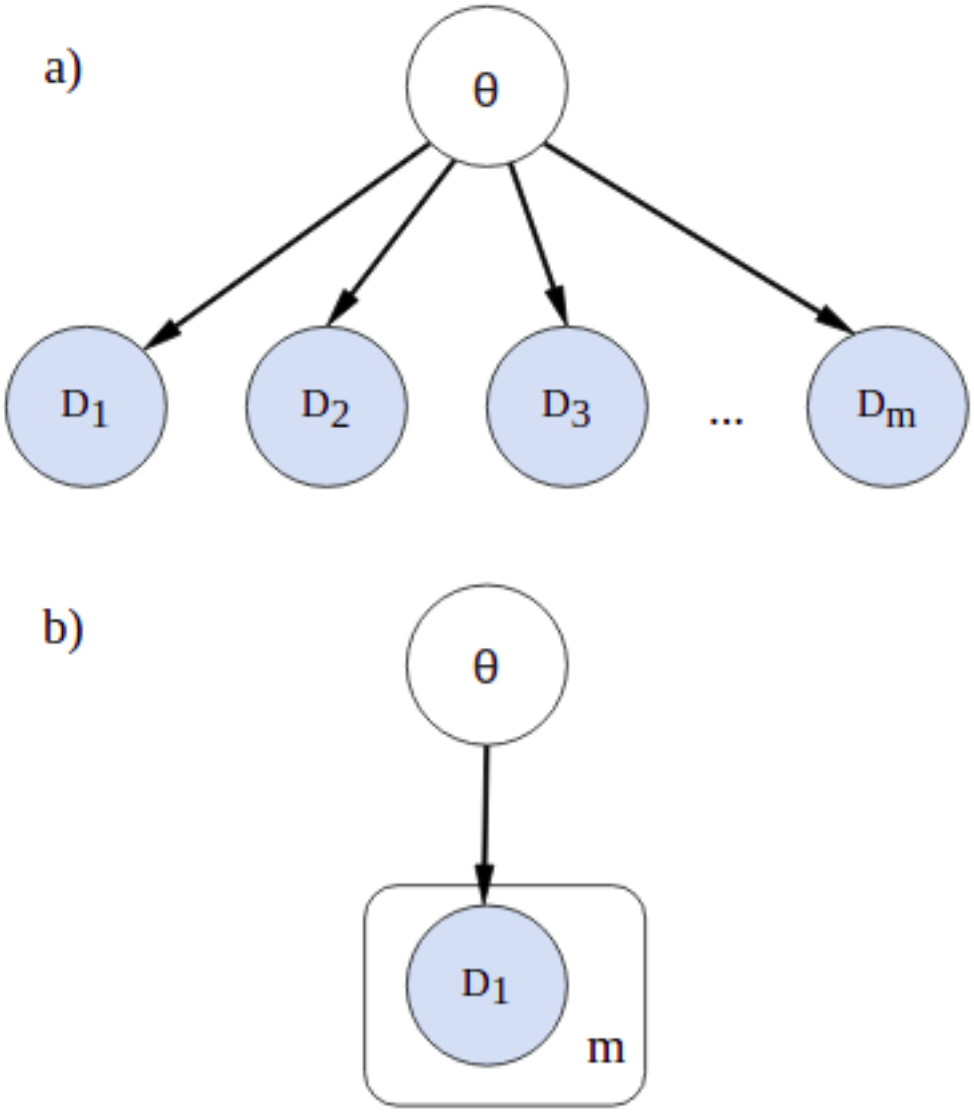


**Figure 30. Bayesian network for learning the bias of a coin in: a) standard notation b) plate notation**

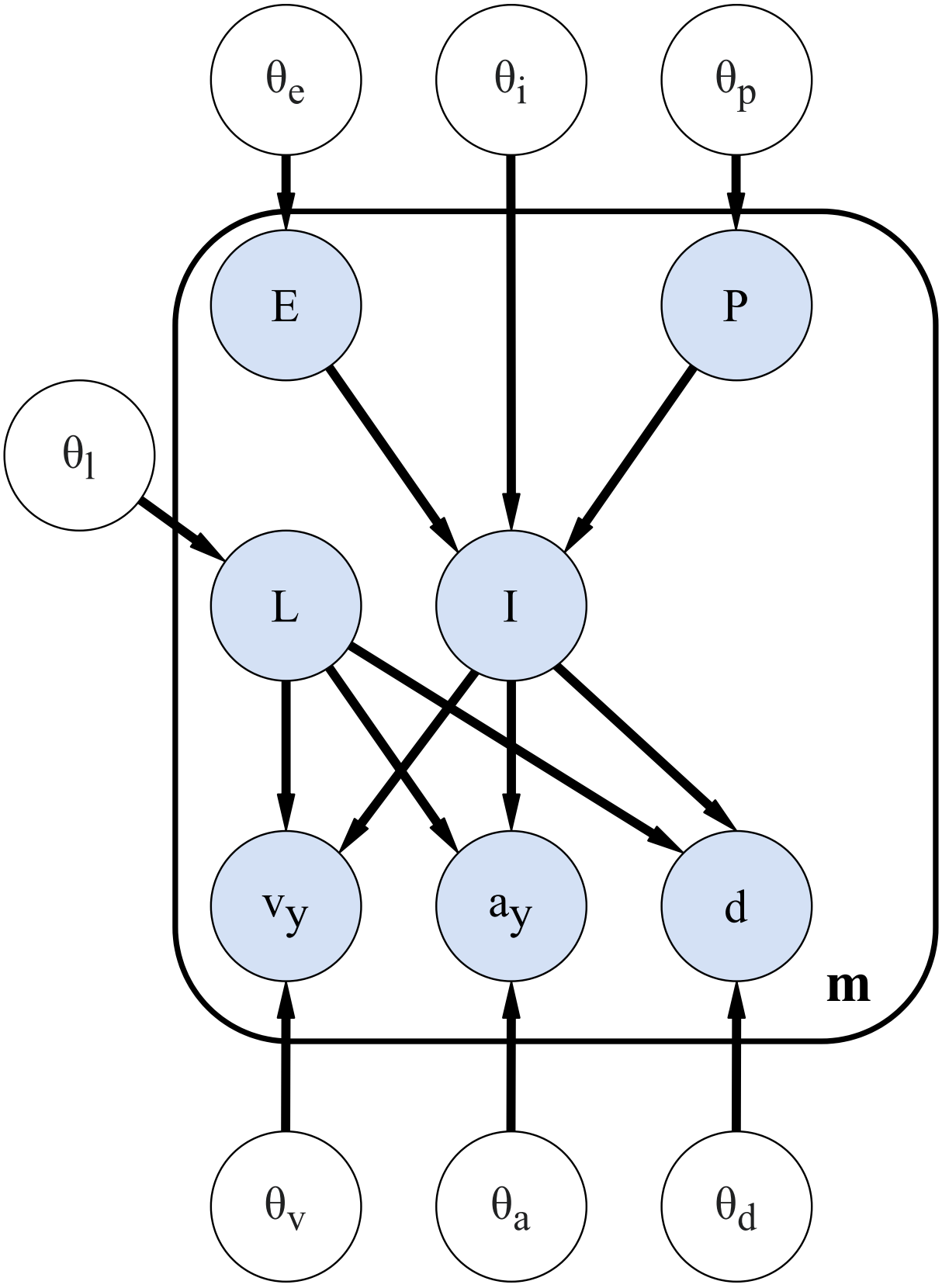


**Figure 31. Network structure**

**Vehicle class** Nodes $E$ and $P$ are representing ego and preceding vehicle class. They are discrete random variables modeled with Bernoulli distribution. Both can take values of *car* or *truck*.

$$\mathrm{E} \sim Bernoulli(\theta_e) \tag{4.1}$$

$$\mathrm{P} \sim Bernoulli(\theta_p) \tag{4.2}$$

The value of parameter intuitively reads as the probability that the vehicle is a car.

**Intention** The node $I$ represents intention random variable. In the learning phase intention is observed value. We assumed lane change intention to be true in a period of four seconds before and after the vehicle touches another lane. Node $I$ is a discrete random variable with values of *True* and *False*.

$$I \sim Bernoulli(\theta_i) \tag{4.3}$$

The parameter $\theta_i$ is here probability that there exists an intention to change a lane. From (Figure 31) we see that node $I$ has parents $E$ and $P$. This means that $P(I)$ depends on the values of E and P.

**Lane** The node $L$ is discrete radnom variable with values of *left lane* and *right lane*.

$$L \sim Bernoulli(\theta_l) \tag{4.4}$$

**Lateral kinematics** The nodes $v_y$ and $a_y$ are representing lateral velocity and acceleration respectivly. Both are continuous random variables modeled with the normal distribution.

$$v_y \sim Normal(\mu_v, \sigma_v) \tag{4.5}$$

$$a_y \sim Normal(\mu_a, \sigma_a) \tag{4.6}$$

Both have only discrete parents so the CPD will be a table of Gaussians. We make a conditional table for every $\mu$ and $\sigma$. This way model learns new normal distribution for every new instantiation of L and I.

**The distance headaway** The node $d$ represents the distance headway (dhw) random variable. Dhw is the distance between the ego vehicle and the preceding vehicle. This variable is the interaction variable which means it takes into account, other participants. The

node $d$ is conitionous variable with HalfNormal distribution, which is normal distribution with mean 0 and all with values positive.

$$d \sim HalfNormal(\theta_d) \tag{4.7}$$

The parameter $\theta_d$ is taken from CPD similary to parameters for normal distributions in $v_y$ and $a_y$.

### 4.1.2 Learning

We used Stochastic variational inference implemented in the pyro framework and explained in 2.4. To achieve this we need to specify the target and variational distribution. We create target distribution as a function `model()`. The variational distribution (function `guide()`)is created with class `AutoGuide()`. This class takes `model` as an input and creates mean-field variational distribution according to that model.

We use `SVI` class for training. To instantiate this class, we need to specify optimizer and loss function. We used Adam optimizer for this implementation for which we needed to specify the learning rate. In the class `Trace_ELBO()` the elbo loss function and method for calculating gradients is implemented. The trace implementation of directed acyclic graphs is explained in section 2.5.1. The following is a standard inference procedure with `SVI` class:

```
svi = SVI(model,guide,optimizer,loss=Trace_ELBO())
for step in range(num_steps):
    svi.step(data)
```

**Figure 32. Standard learning procedure with SVI class**

Method `.step()` here calculates gradients and perform one step of the optimization. The variable `data` is a torch tensor which contains the training data. This variable is passed to model to observe sample sites which we are conditioning on.

Figure 33 shows convergance of negative ELBO loss function.

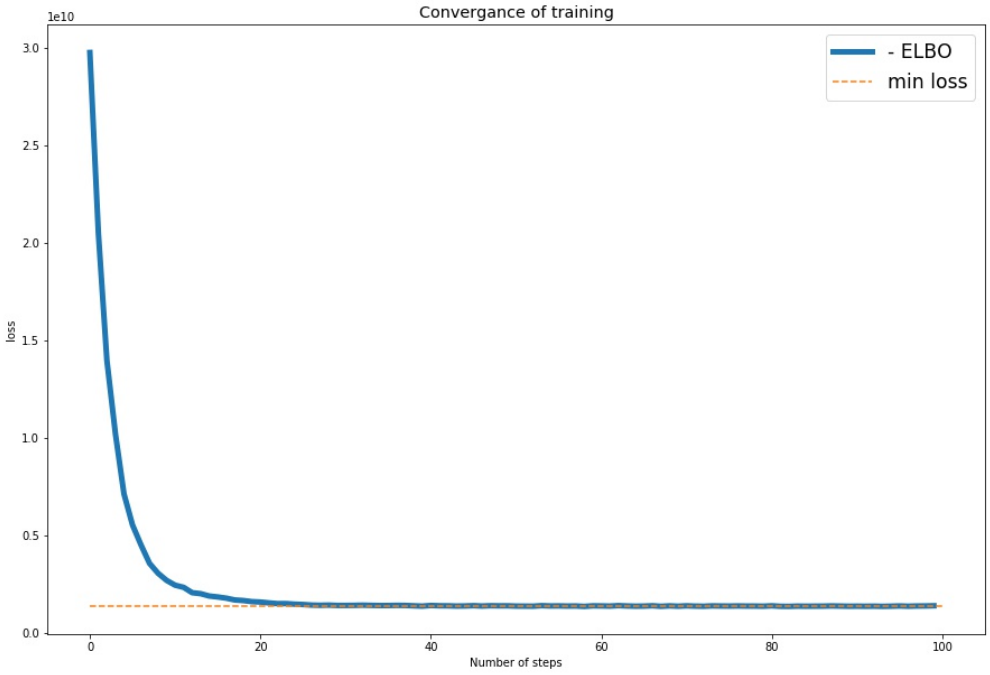


**Figure 33. Change of negative ELBO during learning steps**

The process of learning transforms prior JPD to posterior. If we generate data from prior and posterior distributions we can see what did model learn (Figure 34). This is one of the advantages of probabilistic models and can be used for debugging models.

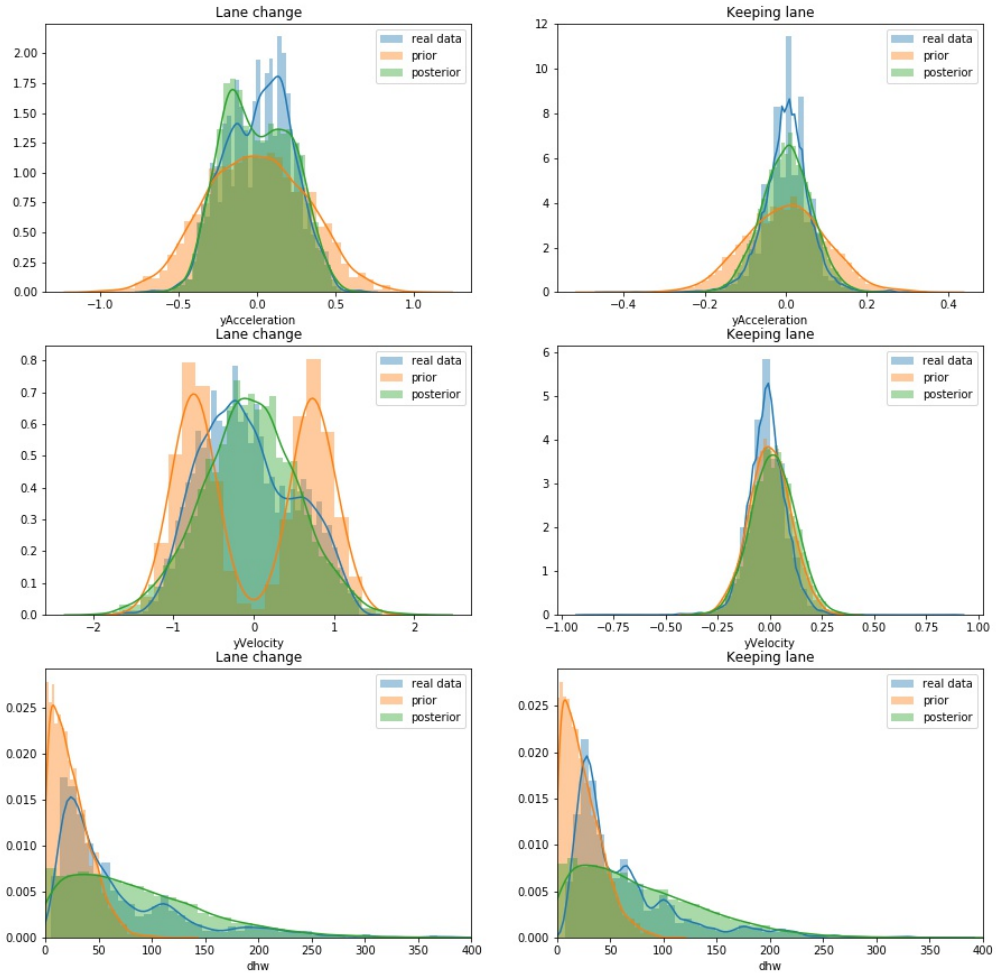


**Figure 34. Comparison of data generated from prior and posterior distribution**

## 4.2 Infering Lane Change Intention

We will use the trained network for predicting lane change intention in test data. Here we will test our assumption that intention can be inferred from vehicle behavior and interaction data. We will observe every variable in our network except variable $I$ which represents lane change intention. Since we have only one discrete latent variable we can

use a simpler inference algorithm for inference. Here we will use `TraceEnumELBO()` class which has `compute_marginals()` method.

We can visualize the probability of lane change intention on sample trajectory from the test dataset (Figure 35). Let us choose one trajectory and plot it together with continuous random variables observed in the network.

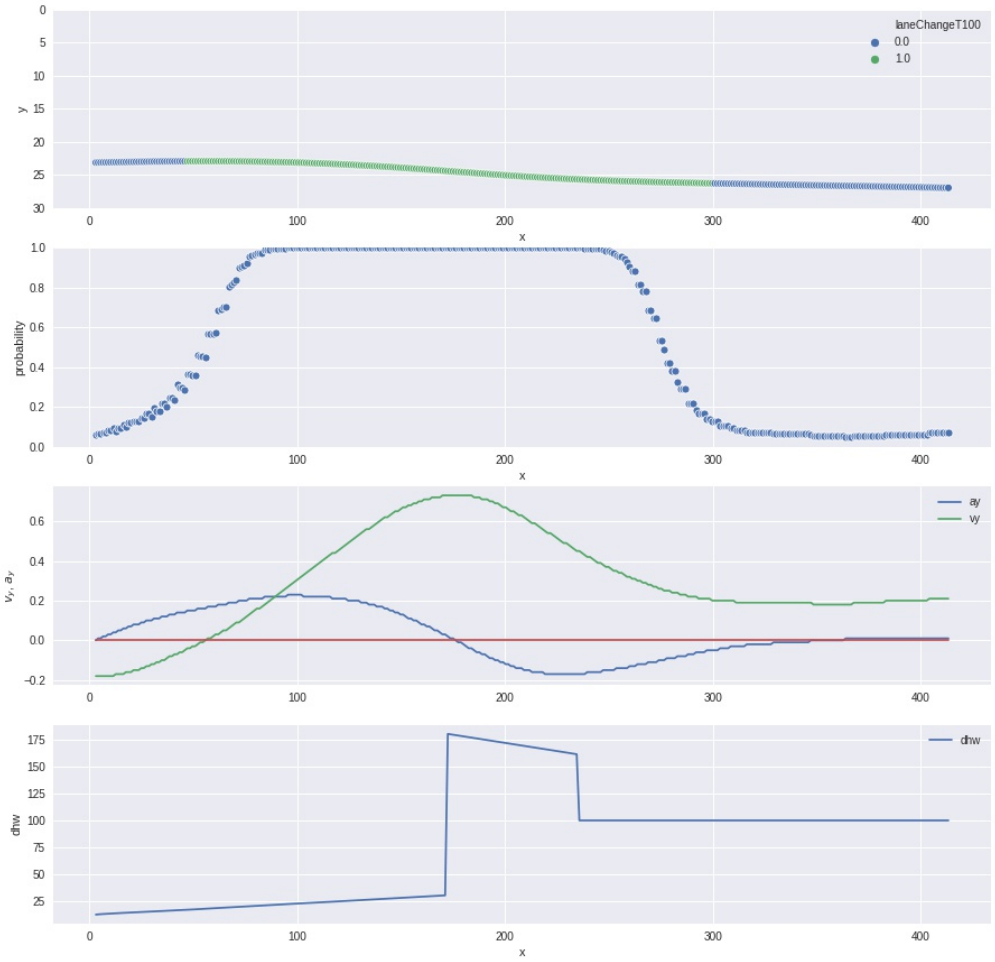


**Figure 35. Prediction visualization for trajectory from test set.**

# 5 Results

One of the most important parts of creating a model is its evaluation. We should know how the model performs on data it has never seen. Moreover, we need a systematic way of evaluating models so we can compare different models. Finding the best model is an iterative procedure that consists of training the model, predicting, and comparing models. To compare models we need to use metrics that can tell us something about the model performance.

Choosing metrics to evaluate the model depends on what problem is being solved. For example, we use different evaluation metrics for regression and classification problems. There are standard metrics that work on most of the problems. These metrics are accuracy, classification error, etc.

Some metrics can be misleading if the dataset is imbalanced. An imbalanced dataset has one label that is dominating in the dataset. If we look at our test dataset we can see that only 13.7% of lane change labels are true. This makes our dataset imbalanced and we should choose metrics accordingly.

## 5.1 Defining evaluation metrics

For evaluating model we need to compare prediction values with actual values in the dataset. Maybe the most used metric in evaluating prediction models is accuracy. Accuracy is defined as:

$$\text{Accuracy} = \frac{\text{Correct predictions}}{\text{Total predictions}}. \tag{5.1}$$

Even though it is often used it is inappropriate for imbalanced sets. To understand this we can imagine a model that would predict lane change to be false for any input. If we have a small number of positive labels for lane change accuracy for this model misleadingly present this model as good.

A lot of metrics are derived from the so-called confusion matrix. A confusion matrix is a tabular presentation of values that we get when we compare predictions with actual data. For binary classification, there are four possible combinations when evaluating predictions. The prediction can be 0 or 1 and the real class can be 0 or 1. These combinations are presented in table 3. Table 3 is called confusion matrix.

**Table 3. Confusion matrix for binary classification**

| | Positive predictions | Negative Predictions |
|---|---|---|
| Positive class | True Positive (TP) | False Negative (FN) |
| Negative class | False Positive (FP) | True Negative (TN) |

We can derive some metrics from the confusion matrix. One such metric tells us what ratio of positive predictions is also positive in the real dataset. This metric is called Precision:

$$\text{Precision} = \frac{TP}{TP + FP}. \tag{5.2}$$

Another metric tells us what ratio of total positive classes in the real dataset is predicted as positive. This metric is called Recall:

$$\text{Recall} = \frac{TP}{TP + FN}. \tag{5.3}$$

There is a motivation for having one metric when evaluating models. Having one metric makes the comparison of models easier. Precision and recall can be joined in one metric. This metric is called F-score (or F1-score):

$$\text{F} - \text{score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{5.4}$$

Precision-Recall (PR) curve is a good measure for comparing models. We can construct the Precision-Recall curve by plotting the curve with Recall on the x-axis and Precision on the y-axis. The ideal model would have a point on (1,1) coordinate on the PR curve. Good models have a curve that is bowing towards (1,1) coordinate. The worst model creates a flat line with a precision that is a proportion of positive class in a dataset.

To present the PR curve as a single metric we use the PR Area Under Curve (AUC) score. The score of 1 represents the ideal model. We can use this score for easier model comparison.

## 5.2 Model Evaluation

With the evaluation metric defined we can evaluate our model. We will evaluate the model for previously described metrics and comment on them. Let us first show confusion matrix (Table 4).

**Table 4. Confusion matrix for model predictions**

| | Positive predictions | Negative Predictions |
|---|---|---|
| Positive class | 14441 | 3206 |
| Negative class | 3768 | 44337 |

Using values from confusion matrix we can calculate precision, recall and f1-score as defined in (5.2), (5.3) and (5.4). The values of these metrics for the positive class are presented in table 6. We can also construct a precision-recall curve (Figure 36) and

**Table 5. Normalized confusion matrix**

| | Positive predictions | Negative Predictions |
|---|---|---|
| Positive class | 21,95% | 4,88% |
| Negative class | 5,72% | 21,95% |

**Table 6. Evaluation metrics of a model**

| Metric | Value |
|---|---|
| Precision | 0,79 |
| Recall | 0,82 |
| f1-score | 0,81 |
| Accuracy | 0,89 |
| PR AUC | 0,8949 |

calculate the area under the curve. This metric is presented in table 6 as PR AUC (Precision-Recall Area Under a Curve).
Another important metric is prediction time. Predictions are not useful if it can not predict lane change early enough. By evaluating on test set this model has a prediction time of 2,5 sec on average. This means that on average the model can recognize lane change intention approximately 2,5 seconds before the vehicle touches another lane.

For the possible implementation of this model in production, the model must predict the intention in real-time. The inference time of a trained network is appropriate for that.

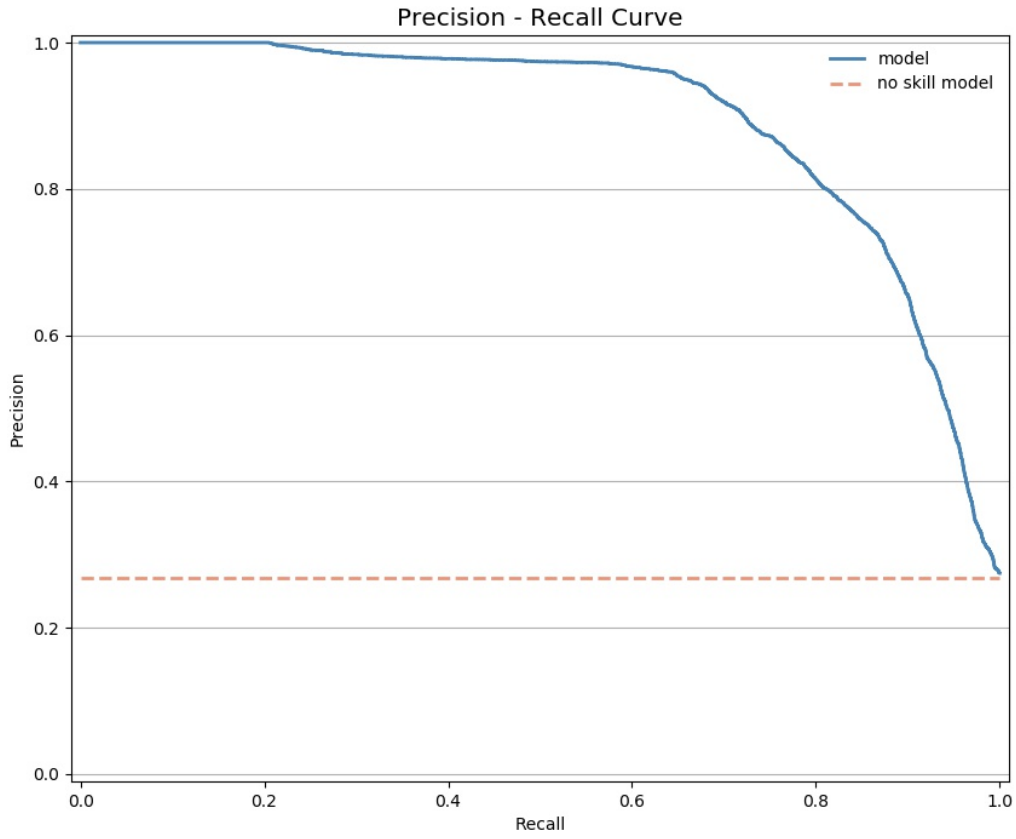


**Figure 36. Precision recall curve**

# 6 Conclusion

Even though fatal traffic accidents are in decline there is still much room for progress. One type of traffic accident occurs when a vehicle is doing a lane-change maneuver and other participants do not notice it. The goal of this thesis was to develop a simple and interpretable model which could predict lane change intentions of highway drivers using Bayesian machine learning methods. This way drivers could be warned about sudden lane change according to model predictions.

The traffic domain is a high-risk domain. The model performing in this domain should be interpretable. They also should have good uncertainty representation on different levels because uncertainty should be taken into account when decisions are made. This is why a Bayesian modeling approach is taken in this thesis.

Writing probabilistic models is hard. Until recently only experts could write even simple probabilistic models. Also, inference in these models is hard and experts had to write a special inference algorithm for every variation of a model. This was a problem because creating a good model is an iterative process.

This thesis leverages probabilistic programming and stochastic variational inference for creating models and doing inference. Probabilistic programming is a way of automating bayesian modeling and inference. To avoid creating new inference algorithms for every new model we use BBVI as implemented in the pyro programming language. This makes combining continuous and discrete nodes simple and also experimenting with different models easy. Stochastic optimization in BBVI makes scaling to large datasets possible.

The model created in this thesis is a hybrid Bayesian network with three continuous and four discrete nodes. All discrete nodes are Bernoulli distributions and all continuos nodes are Normal distributions. One can try various distributions for modeling continuous variables but Normal distributions gave the best results in this case. The model was trained and tested on a modified highD dataset. The evaluation shows good predicting power comparable with other Bayesian network approaches. The model can recognize lane change maneuver intention with an F1-score of 81%.

Data collected in highD dataset is recorded with a drone from a bird's perspective. The data that vehicles collect in real-life situations is different. This is one limitation of this model. Another limitation is that model is created for a highway with two lanes. The model is static. It takes only information for one frame at the time and is indifferent to the time dimension.

The power of probabilistic programming can be used for extending this model. One obvious extension is to create a model that works on a highway with an arbitrary number of lanes. The model can be extended by adding more nodes and dependencies in its structure. Some of these can be longitudinal kinematics, distance from lane markings, information

about other lanes, interaction with other participants besides preceding vehicle, etc. This model can also be extended to capture temporal dependencies to the so-called dynamic bayesian network. One can assume that model would benefit from information from the past influencing the prediction.

To extend the model even more we can use the fact that pyro is a python framework and to utilize python flexibility while creating a model. Concretely we can create a network with a dynamic structure. This is not the same as Dynamic Bayesian Network, rather it means that the number of nodes in the network can be changed during inference this way. The structure of the network can depend on the number of vehicles in a scene for example.

This thesis showed how a simple Bayesian model can be used for predicting lane changes. Results from this thesis can be used as an introduction to modeling lane changes with probabilistic programming. Future research could be about network extensions and solving previously stated limitations to create a production-ready model.

# References


[1] European Commision. "Saving Lives: Boosting Car Safety in the EU". In: (2016).

[2] Karel Brookhuis, Dick de Waard, and Wiel Janssen. "Behavioural impacts of Advanced Driver Assistance Systems - an overview". In: *European Journal of Transport and Infrastructure Research* 1.3 (2001). ISSN: 1567-7141. DOI: 10.18757/ejtir.2001.1.3.3667.

[3] Mohamed Shawky. "Factors affecting lane change crashes". In: *IATSS Research* (2020). ISSN: 0386-1112. DOI: https://doi.org/10.1016/j.iatssr.2019.12.002. URL: http://www.sciencedirect.com/science/article/pii/S0386111219300020.

[4] Thomas Dingus et al. "The 100-Car Naturalistic Driving Study: Phase II - Results of the 100-Car Field Experiment". In: (Jan. 2006).

[5] Jan-Willem van de Meent et al. *An Introduction to Probabilistic Programming*. 2018. arXiv: 1809.10756 [stat.ML].

[6] Jonas Firl. "Probabilistic Maneuver Recognition in Traffic Scenarios". PhD thesis. 2014. 150 pp. ISBN: 978-3-7315-0287-6. DOI: 10.5445/KSP/1000043680.

[7] Z. Ghahramani. "Probabilistic machine learning and artificial intelligence". In: *Nature, 521(7553)* (2015), pp. 452–459.

[8] Li Junxiang et al. "A Dynamic Bayesian Network for Vehicle Maneuver Prediction in Highway Driving Scenarios: Framework and Verification". In: *Electronics* 8 (Jan. 2019), p. 40. DOI: 10.3390/electronics8010040.

[9] D. Lee et al. "Convolution neural network-based lane change intention prediction of surrounding vehicles for ACC". In: *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*. 2017, pp. 1–6.

[10] D. J. Phillips, T. A. Wheeler, and M. J. Kochenderfer. "Generalizable intention prediction of human drivers at intersections". In: *2017 IEEE Intelligent Vehicles Symposium (IV)*. 2017, pp. 1665–1670.

[11] D. Kasper et al. "Object-oriented Bayesian networks for detection of lane change maneuvers". In: *2011 IEEE Intelligent Vehicles Symposium (IV)*. 2011, pp. 673–678.

[12] T. Gindele, S. Brechtel, and R. Dillmann. "Learning Driver Behavior Models from Traffic Observations for Decision Making and Planning". In: *IEEE Intelligent Transportation Systems Magazine* 7.1 (2015), pp. 69–79.

[13] R. Schubert and G. Wanielik. "A Unified Bayesian Approach for Object and Situation Assessment". In: *IEEE Intelligent Transportation Systems Magazine* 3.2 (2011), pp. 6–19.

[14] M. Liebner et al. "Driver intent inference at urban intersections using the intelligent driver model". In: *2012 IEEE Intelligent Vehicles Symposium*. 2012, pp. 1162–1167.

[15] Alan Turing. "Computing Machinery and Intelligence". In: *Mind, a Quarterly Review of Psychology and Philosophy* 59.236 (1950).

[16] Korb Nicholson. *Bayesian Artifitial Inteligence*. Chapman Hall, 2004.

[17] Judea Pearl. *From Bayesian Networks to Causal Networks*. 1995.

[18] A.J. Bondy and U.S.R. Murty. *Graph Theory with Applications*. Wiley, 1991. ISBN: 9780471363248. URL: https://books.google.de/books?id=7EWKkgEACAAJ.

[19] Ben-Gal I. "Bayesian Networks". In: *Encyclopedia of Statistics in Quality Reliability* (2007).

[20] Rajesh Ranganath, Sean Gerrish, and David M. Blei. "Black Box Variational Inference". In: *ArXiv* abs/1401.0118 (2014).

[21] F.M Dekking. *A Modern Introduction to Probability and Statistics*. Springer, 2005.

[22] *Oreilly - Hands on cnn's*. https://www.oreilly.com/library/view/hands-on-convolutional-neural/9781789130331/1d8faa08-c404-4c12-929d-711b2cd995d6.xhtml.

[23] E. Cinlar. *Probability and Stochastics*. Springer, 2011.

[24] Eli Bingham et al. "Pyro: Deep Universal Probabilistic Programming". In: *ArXiv* abs/1810.09538 (2019).

[25] David Wingate and Teophane Webber. "Automated Variational Inference in Probabilistic Programming". In: *ArXiv* (2013).

[26] Robert Krajevski et al. "The highD Dataset: A Drone of Naturalistic Vehicle Trajectories on German Highways for Validation of Highly Automated Vehicles". In: *2018 21st International Conference on Intelligent Transportation Systems (ITSC*. 2018.

# Appendix

## model_utils.py

```
import pandas as pd
import torch

PATH_TRAIN = "../data/interim/data_v1.csv"


def load_data(PATH, tensor = False, drop = None):
    df = pd.read_csv(PATH)
    if drop is not None:
        df = df.drop(
            columns= drop
        )

    return df

def load_data_as_tensor(PATH, drop = None):
    df = pd.read_csv(PATH)

    if drop is not None:
        df = df.drop(
            columns= drop
        )

    return torch.tensor(df.values)
```

## model4$_v$2.*py*

```
import torch
import pyro
import torch.distributions.constraints as constraints
from pyro import distributions as dist
from pyro.contrib.autoguide import AutoGuide


class Model4:
    def model(self, data):
```

```
ego_class_loc = pyro.param("ego_loc", torch.tensor(0.5),
   constraint=constraints.unit_interval)
preceding_class_loc = pyro.param("prec_loc",
   torch.tensor(0.5),
   constraint=constraints.unit_interval)

intention = pyro.param(
    "intention_cpd",
    torch.tensor([[0.04, 0.06], [0.2, 0.15]]),
    constraint=constraints.unit_interval,
)
a_loc = pyro.param("a_loc_p", torch.tensor([[0.0, 0.0],
   [0.2, -0.2]]))
a_scale = pyro.param(
    "a_scale_p",
    torch.tensor([[0.1, 0.1], [0.25, 0.25]]),
    constraint=constraints.positive,
)
v_loc = pyro.param("v_loc_p", torch.tensor([[0.0, 0.0],
   [0.75, -0.75]]))
v_scale = pyro.param(
    "v_scale_p",
    torch.tensor([[0.1, 0.1], [0.25, 0.25]]),
    constraint=constraints.positive,
)
dhw_param_a = pyro.param(
    "dhw_param_a",
    torch.tensor([[3, 3], [3, 3]], dtype=float),
    constraint=constraints.positive,
)


with pyro.plate("data", data.shape[0], dim=-2,
   subsample_size=500) as idx:
    ego_class = pyro.sample(
        "ego_class", dist.Bernoulli(ego_class_loc),
           obs=data[idx, 4]
    )
    preceding_class = pyro.sample(
        "preceding_class",
```

```
                dist.Bernoulli(preceding_class_loc),
                obs=data[idx, 5]
            )

            pa_intention = [ego_class.long(),
               preceding_class.long()]
            lane_change = pyro.sample(
                "lane_change",
                dist.Bernoulli(intention[pa_intention]),
                obs=data[idx, 0]
            )

            lane = pyro.sample("lane", dist.Bernoulli(0.5),
               obs=data[idx, 1])

            parents = [lane_change.long(), lane.long()]

            pyro.sample("dhw",
               dist.HalfNormal(dhw_param_a[parents]),
               obs=data[idx, 6])
            pyro.sample(
                "a", dist.Normal(a_loc[parents],
                a_scale[parents]), obs=data[idx, 2]
            )
            pyro.sample(
                "v", dist.Normal(v_loc[parents],
                v_scale[parents]), obs=data[idx, 3]
            )

    def guide(self, data):
        return AutoGuide(self.model)
```

### train.py

```
import argparse

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pyro
```

```
import pyro.distributions as dist
import torch
from pyro import optim
from pyro.infer import SVI, Trace_ELBO

from model_utils import PATH_TRAIN, load_data_as_tensor
from model4_v2 import Model4 as Model


def train(model, guide, data, steps, save_every, lr):
    pyro.enable_validation(True)

    Elbo = Trace_ELBO
    elbo = Elbo(5)
    adam = optim.Adam({"lr": lr})

    svi = SVI(model, guide, adam, elbo)

    pyro.get_param_store().clear()
    loss_history = []
    for step in range(1,steps+1):
        loss = svi.step(data)
        if step % 1 == 0:
            print("step {} loss = {}".format(step, loss))
            loss_history.append(loss)

        if step % save_every == 0:
            pyro.get_param_store().save("models/saved_models/T100_{}_{}_{}_n

    print("Training done!")
    return loss_history

def main():
    parser = argparse.ArgumentParser(description="Train the
       model")
    parser.add_argument(
        "--num_steps", type=int, default=1000, help="Number of
           steps",
    )
    parser.add_argument(
```

```
        "--save_every", default=10000, type=int, help="Save
            weights every n steps ",
    )
    parser.add_argument(
        "--learning_rate", type=float, default=1e-2,
            help="Learning rate for ADAM optimizer",
    )

    data =
        load_data_as_tensor("../data/interim/data_2lane_sym_balanced_wprec_2wa
        drop="sceneId")
    data = data[:,2:]
    data = data[:,[56,60,5,3,13,17,6]]

    args = parser.parse_args()
    BN = Model()


    loss = train(BN.model, BN.guide, data, args.num_steps,
        args.save_every, args.learning_rate)

    plt.plot(loss)
    plt.savefig("../reports/loss function.jpg")
    plt.show()

    df_loss = pd.DataFrame({"loss": loss}, index=None)
    df_loss.to_csv("loss_history_4v2_{}_2way.csv".format(args.learning_rate)


if __name__ == "__main__":
    main()
```

## balance_data.py

```
import pandas as pd
import random


def has_preceding(df) -> pd.DataFrame:
    return df[df["precedingId"] != 0]
```

```
def is_at_least_one_change(numLaneChanges: pd.Series) ->
   pd.Series:
    return numLaneChanges.map(lambda x: 1 if x > 0 else 0)


def get_change_ids(meta: pd.DataFrame, laneChange: pd.Series) ->
   list:
    return meta[laneChange == 1].id.values.tolist()


def get_keep_ids(meta: pd.DataFrame, laneChange: pd.Series) ->
   list:
    return meta[laneChange == 0].id.values.tolist()


def sample_keep_ids(keep_ids: list, num_ids: int) -> list:
    return random.sample(keep_ids, num_ids)


def get_id_data(df: pd.DataFrame, keep_ids: list) -> list:
    dfs = []
    for idx in keep_ids:
        dfs.append(df[df["id"] == idx])
    return dfs


def create_final_id_vector(keep_ids: list, change_ids: list) ->
   list:
    final = keep_ids + change_ids
    random.shuffle(final)
    return final


def concat_data(dfs: list) -> pd.DataFrame:
    return pd.concat(dfs, axis=0)


def balance_data(df: pd.DataFrame, meta: pd.DataFrame) -> tuple:
```

```
    """Choose vehicles so that there is similar number of
       vehicles that
    changed lane, and ones that stayed in the lane

    Arguments:
        df {[type]} -- [description]
        meta {[type]} -- [description]

    Returns:
        tuple -- [description]
    """
    df = has_preceding(df)
    laneChange = is_at_least_one_change(meta.numLaneChanges)
    change_ids = get_change_ids(meta, laneChange)
    keep_ids = get_keep_ids(meta, laneChange)
    num_change_ids = len(change_ids)
    sampled_keep_ids = sample_keep_ids(keep_ids, num_change_ids)
    final_ids = create_final_id_vector(sampled_keep_ids,
       change_ids)
    dfs = get_id_data(df, final_ids)
    df = concat_data(dfs)
    msg = [
        " Balanced data so that there is {} vehicles that
           changed \
            lane and {} vehicles that stayed in the lane and
               that every ego has preceding car".format(
            num_change_ids, len(sampled_keep_ids)
        )
    ]
    return df, msg
```

## balance_data_test.py

```
import unittest
import pandas as pd
import balance_data as bd

meta = pd.DataFrame()
meta["numLaneChanges"] = [0, 0, 0, 1, 2]
meta["id"] = [1, 2, 3, 4, 5]
```

```
df = pd.DataFrame()
df["id"] = [1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 5,
   5, 5, 5]
df["precedingId"] = [1, 0, 2, 0, 3, 0, 4, 0, 1, 0, 1, 0, 1, 0,
   1, 0, 1, 0, 1, 0]

class TestBalancingData(unittest.TestCase):
    def test_has_preceding(self):
        out = bd.has_preceding(df)
        print(out)
        self.assertFalse(0 in out.precedingId.values.tolist())


    def test_is_one_change(self):
        out = bd.is_at_least_one_change(meta.numLaneChanges)
        self.assertTrue(isinstance(out, pd.Series))
        self.assertListEqual(out.values.tolist(), [0, 0, 0, 1,
           1])

    def test_get_change_ids(self):
        is_one = bd.is_at_least_one_change(meta.numLaneChanges)
        out = bd.get_change_ids(meta, is_one)

        self.assertTrue(isinstance(out, list))
        self.assertListEqual(out, [4, 5])

    def test_get_keep_ids(self):
        is_one = bd.is_at_least_one_change(meta.numLaneChanges)
        out = bd.get_keep_ids(meta, is_one)

        self.assertTrue(isinstance(out, list))
        self.assertListEqual(out, [1, 2, 3])

    def test_sample_keep_ids(self):
        self.assertTrue(isinstance(bd.sample_keep_ids([1, 2, 3,
           4, 5], 2), list))
        self.assertEqual(len(bd.sample_keep_ids([1, 2, 3, 4, 5],
           2)), 2)
```

```
    def test_create_final(self):
        self.assertEqual(len(bd.create_final_id_vector([1,2],[4,5])),4)
        self.assertTrue(1 in
           bd.create_final_id_vector([1,2],[4,5]))
        self.assertTrue(2 in
           bd.create_final_id_vector([1,2],[4,5]))
        self.assertTrue(4 in
           bd.create_final_id_vector([1,2],[4,5]))
        self.assertTrue(5 in
           bd.create_final_id_vector([1,2],[4,5]))


    def test_get_id_data(self):
        out = bd.get_id_data(df, [1, 3])
        self.assertTrue(isinstance(out, list))
        self.assertEqual(len(out), 2)

    def test_concat_data(self):
        dfs = bd.get_id_data(df, [1, 3,4,5])
        out = bd.concat_data(dfs)

        self.assertEqual(
            out.id.values.tolist(), [1, 1, 1, 1, 3, 3, 3, 3, 4,
               4, 4, 4, 5, 5, 5, 5]
        )

    def test_balance_data(self):
        out,_ = bd.balance_data(df,meta)

        self.assertTrue(isinstance(out,pd.DataFrame))


if __name__ == "__main__":
    unittest.main()
```

## join_dataframes.py

```
import pandas as pd

scenes = [ "01",  "02",  "03", "15", "16", "17", "18", "19",
```

```
    "20", "21", "22", "23"]
dfs = []
for scene in scenes:
    path =
        "../data/interim/data_scene_{}_sym_balanced_wprec_2way.csv".format(sce
    df = pd.read_csv(path)
    dfs.append(df)

df_final = pd.concat(dfs, axis=0)
df_final.to_csv("../data/interim/data_2lane_sym_balanced_wprec_2way.csv")
```

### lane_changes.py

```
import pandas as pd
import numpy as np


def lane_change_categorize(lane_changes: pd.Series) -> pd.Series:
    """Change number of lane changes collumn to 0 if there is no
       lane change and 1 if there is 1 or more lane changes"""
    return lane_changes.map(lambda x: 0 if x == 0 else 1)


def id_laneChange(meta: pd.DataFrame) -> dict:
    """Create mapp of vehicles and action regarding lane change

    Arguments:
        meta {pd.DataFrame}

    Returns:
        dict
    """
    return dict(zip(meta.id.values,
       lane_change_categorize(meta.numLaneChanges).values))

def end_trajectory(id_same: list) -> list:
    i = 0
    until_end = []
    for l in reversed(id_same):
        if l == 1:
```

```
            i += 1
        elif l == 0:
            i=0
        until_end.append(i)
    return list(reversed(until_end))


def begining_trajectoy(id_same: list) -> list:
    i = 0
    until_begin = []
    for l in id_same:
        if l == 1:
            i += 1
        elif l == 0:
            i=0
        until_begin.append(i)
    return until_begin


def add_lane_change(tracks, meta) -> (pd.DataFrame, list):
    msg = " Add laneChange collumn to dataframe "
    tracks["laneChange"] = tracks.id.map(id_laneChange(meta))
    return tracks, [msg]


def same_check(vector: list) -> list:
    return [ x==x1 for x,x1 in zip(vector[:-1],vector[1:])]

def different_check(vector: list) -> list:
    return [ x!=x1 for x,x1 in zip(vector[:-1],vector[1:])]

def expand_label_past(labels: list, num_frames: int) -> list:
    expanded_labels = np.zeros_like(labels).tolist()
    true_sublist = np.ones(num_frames+1)
    for idx,label in enumerate(labels):
        if label:
            expanded_labels[idx-num_frames:idx+1] = true_sublist
    return expanded_labels
```

```
def expand_label_past_future(labels: list, num_frames: int,
   to_end:list, to_begining) -> list:
    expanded_labels = np.zeros_like(labels).tolist()
    for idx,label in enumerate(labels):
        if label:
            lim_begin = np.min([num_frames,to_end[idx]])
            lim_end = np.min([num_frames,to_begining[idx]])
            sublist = np.ones([lim_begin+lim_end +1])
            expanded_labels[idx-lim_begin:idx+lim_end+1] =
               sublist
    return expanded_labels

def compare_lists(list1:list,list2:list) -> list:
    array1 = np.array(list1)
    array2 = np.array(list2)
    compared = array1 & array2
    compared = list(map(int,compared))
    return compared

def add_lane_change_t(df, num_frames: int) -> tuple:
    """[summary]

    Arguments:
        tracks {[type]} -- original dataframe
        num_frames {int} -- how long do we want to expand label
           on the trajectory

    Returns:
        tuple -- (new dataframe, message)
    """
    is_id_same = same_check(df.id.values.tolist())
    is_lane_same = different_check(df.laneId.values.tolist())
    is_lanechange = compare_lists(is_id_same,is_lane_same)
    to_end = end_trajectory(is_id_same)
    to_begining = begining_trajectoy(is_id_same)

    labeled_lanechanges =
       expand_label_past_future(is_lanechange,
       num_frames,to_end,to_begining)
```

```
    column_name = "laneChangeT{}".format(num_frames)
    msg = [" Add {} to dataset".format(column_name)]
    reshaped_labeled_lanechanges =  [0]
    reshaped_labeled_lanechanges.extend(labeled_lanechanges)
    df[column_name] = reshaped_labeled_lanechanges
    return df, msg
```

### lane_changesq_test.py

```
import unittest
import pandas as pd
import numpy as np
import lane_changes as lc


class TestLaneChangeLabeling(unittest.TestCase):
    def test_same_check(self):
        self.assertEqual(lc.same_check([1, 1, 1, 1, 1, 1]), [1,
           1, 1, 1, 1])
        self.assertEqual(lc.same_check([1, 1, 1, 2, 2, 2]), [1,
           1, 0, 1, 1])

    def test_different_check(self):
        self.assertEqual(lc.different_check([1, 1, 1, 1, 1, 1]),
           [0, 0, 0, 0, 0])
        self.assertEqual(lc.different_check([1, 1, 1, 2, 2, 2]),
           [0, 0, 1, 0, 0])

    def test_end(self):
        self.assertEqual(
            lc.end_trajectory([1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1,
               1, 0, 1, 1, 1]),
            [6, 5, 4, 3, 2, 1, 0, 5, 4, 3, 2, 1, 0, 3, 2, 1],
        )

    def test_begin(self):
        self.assertEqual(
            lc.begining_trajectoy([1, 1, 1, 1, 1, 1, 0, 1, 1, 1,
               1, 1, 0, 1, 1, 1]),
            [1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 0, 1, 2, 3],
```

```
        )

    def test_expand_label_past(self):
        self.assertEqual(
            lc.expand_label_past([0, 0, 0, 1, 0, 0, 0], 2), [0,
                1, 1, 1, 0, 0, 0]
        )
        self.assertEqual(
            lc.expand_label_past([0, 0, 0, 1, 0, 0, 0], 2), [0,
                1, 1, 1, 0, 0, 0]
        )

    def test_expand_label_past_future(self):
        self.assertEqual(
            lc.expand_label_past_future(
                [0, 0, 0, 1, 0, 0, 0], 2, [6, 5, 4, 3, 2, 1, 0],
                    [0, 1, 2, 3, 4, 5, 6]
            ),
            [0, 1, 1, 1, 1, 1, 0],
        )


    def test_compare_lists(self):
        self.assertEqual(lc.compare_lists([1, 1, 0, 0], [1, 0,
            1, 0]), [1, 0, 0, 0])
        self.assertTrue(
            isinstance(lc.compare_lists([1, 1, 0, 0], [1, 0, 1,
                0])[0], int)
        )

    def test_add_lanechange_t(self):
        df = pd.DataFrame(
            {
                "id": [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                "laneId": [1, 1, 1, 1, 1, 2, 2, 2, 2, 2],
            }
        )

        df_new, _ = lc.add_lane_change_t(df, 3)
        self.assertTrue(isinstance(df_new, pd.DataFrame))
```

```
        self.assertEqual(df_new.shape, (10, 3))
        self.assertListEqual(
            df_new.columns.values.tolist(), ["id", "laneId",
               "laneChangeT3"]
        )
        self.assertListEqual(
            df_new.laneChangeT3.values.tolist(), [0, 0, 1, 1, 1,
               1, 1, 1, 1, 0]
        )


if __name__ == "__main__":
    unittest.main()
```

## lane_relations.py

```
import pandas as pd


def get_markings(rec: pd.DataFrame) -> [dict, dict]:
    upper_lane_markings = list(
        map(lambda x: float(x),
            rec.upperLaneMarkings.values[0].split(";"))
    )
    lower_lane_markings = list(
        map(lambda x: float(x),
            rec.lowerLaneMarkings.values[0].split(";"))
    )

    left_mark = {
        2: upper_lane_markings[1],
        3: upper_lane_markings[2],
        5: lower_lane_markings[0],
        6: lower_lane_markings[1],
    }

    right_mark = {
        2: upper_lane_markings[0],
        3: upper_lane_markings[1],
        5: lower_lane_markings[1],
```

```
        6: lower_lane_markings[2],
    }

    return [left_mark, right_mark]


def get_diff(df, mark: dict) -> pd.Series:
    map_mark = df.laneId.map(mark)
    return df.y - map_mark


def add_diff(df, rec):
    msg = []
    marks = get_markings(rec)
    names = ["diffLeft", "diffRight"]

    for name, mark_map in zip(names, marks):
        diff = get_diff(df, mark_map)
        df[name] = diff
        msg.append("Add {} column to dataframe".format(name))

    return df, msg


def categorize_lanes(df: pd.DataFrame) -> pd.DataFrame:
    msg = ["Turning laneId column to 0,1 values where 0 is left
       and 1 is right lane"]
    left_right = {3: 0, 5: 0, 2: 1, 6: 1}

    df["laneIdRL"] = df.laneId.map(left_right)

    return df,msg

def create_height_map(meta:pd.DataFrame) -> dict:
    return dict(zip(meta.id.values,meta.height.values))

def y_to_cm(df:pd.DataFrame,meta:pd.DataFrame) -> tuple:
    """Transform y coordinate from being top left corner of the
       bounding box
    to being coordinate of middle of the box
```

```
    Arguments:
        df {pd.DataFrame} -- original dataframe
        meta {pd.DataFrame} -- meta dataframe

    Returns:
        tuple -- (df:pd.Dataframe, msg:list(str))
    """
    height_vector = df.id.map(create_height_map(meta))
    half_height = height_vector.values / 2
    df["y"] = df.y.values + half_height

    msg = ["Converted y to y_cm"]
    return df, msg
```

### lane_relations_test.py

```
import unittest
import pandas as pd
import lane_relations as lr

meta = pd.DataFrame()
meta["id"] = [1,2,3,4]
meta["height"] = [5,4,3,2]

df = pd.DataFrame()
df["id"] = [1,1,4,4]
df["y"] = [0,0,0,0]

class TestLaneRelations(unittest.TestCase):
    def test_height_map(self):
        self.assertDictEqual(lr.create_height_map(meta),{1:5,2:4,3:3,4:2})

    def test_y_to_cm(self):
        self.assertListEqual(lr.y_to_cm(df,meta)[0].y.values.tolist(),[2.5,2


if __name__ == "__main__":
    unittest.main()
```

## main.py

```
import pandas as pd

from lane_changes import add_lane_change, add_lane_change_t
from lane_relations import add_diff, categorize_lanes, y_to_cm
from vehicle_class import add_class
from relative_veh_relations import add_veh_relations
from reporting_features import create_latex_report
from postprocces import drop_data, normalize_data, add_scene_id
from balance_data import balance_data
from symetry import add_symetry

# scenes = [ "01",  "02",  "03", "15", "16", "17", "18", "19",
   "20", "21", "22", "23", "24"]
scenes = ["24"]
for scene in scenes:
    PATH_META = "../data/raw/{}_tracksMeta.csv".format(scene)
    PATH_TRACKS = "../data/raw/{}_tracks.csv".format(scene)
    PATH_REC = "../data/raw/{}_recordingMeta.csv".format(scene)

    df_tracks = pd.read_csv(PATH_TRACKS)
    df_meta = pd.read_csv(PATH_META)
    df_rec = pd.read_csv(PATH_REC)

    msgs = []

    print("SCENE {}".format(scene))

    df, msg = y_to_cm(df_tracks, df_meta)
    msgs.extend(msg)
    print("y -> y_cm")
    df, msg = add_diff(df, df_rec)
    msgs.extend(msg)
    print("Diff added")
    df, msg = add_class(df, df_meta)
    msgs.extend(msg)
    print("Class added")
    df, msg = add_symetry(df)
    msgs.extend(msg)
```

```
print("Symetry added")
df, msg = add_veh_relations(df)
msgs.extend(msg)
print("Veh relations added")
df, msg = add_lane_change(df,df_meta)
msgs.extend(msg)
print("Lane changes added for whole trajectory")
df, msg = add_lane_change_t(df,200)
msgs.extend(msg)
df, msg = add_lane_change_t(df,150)
msgs.extend(msg)
df, msg = add_lane_change_t(df,100)
msgs.extend(msg)
df, msg = add_lane_change_t(df,75)
msgs.extend(msg)
df, msg = add_lane_change_t(df,50)
msgs.extend(msg)
df, msg = add_lane_change_t(df,25)
msgs.extend(msg)
print("Lane change added")
df, msg = categorize_lanes(df)
msgs.extend(msg)
print("Lanes categorized")
# df, msg = balance_data(df,df_meta)
# msgs.append(msg)
# print("Data balanced")
df, msg = add_scene_id(df,scene)
msgs.append(msg)
print("Scene and id column")
df, msg = drop_data(df)
msgs.append(msg)
print("Droped not needed data")

# df, msg = normalize_data(df)
# msgs.append(msg)




#
```

```
        df.to_csv("../data/interim/data_scene_{}_sym_balanced_wprec_2way.csv"

    # create_latex_report(msgs,
        "../reports/data_scene_{}_sym_balanced_wprec_2way".format(scene))


    df.to_csv("../data/interim/test_data.csv")

    create_latex_report(msgs, "../reports/test_data")

    print("Data created for scene {}".format(scene))
```

### postprocces.py

```
import random
import pandas as pd
from sklearn import preprocessing

COLUMNS_TO_DROP = [
    "id",
    "frame",
    "width",
    "height",
    "frontSightDistance",
    "backSightDistance",
    "ttc",
#     "precedingId",
    "followingId",
    "leftPrecedingId",
    "leftAlongsideId",
    "leftFollowingId",
    "rightPrecedingId",
    "rightAlongsideId",
    "rightFollowingId",
]


def drop_data(df: pd.DataFrame) -> (pd.DataFrame, list):
    msg = ["Droped following columns:
        {}".format(COLUMNS_TO_DROP)]
```

```
    return df.drop(columns=COLUMNS_TO_DROP), msg

def add_scene_id(df:pd.DataFrame, scene) -> tuple:
    df["sceneId"] = ["{}_{}".format(scene,i) for i in df.id]
    msg = ["Added sceneId column"]
    return df, msg

def normalize_data(df):
    cols = list(set(df.columns) - set(["laneId", "vehClasses",
       "laneChange"]))
    msg = "Normalized columns except {}".format(cols)
    scaler = preprocessing.MinMaxScaler()
    scaler.fit(df[cols].values)
    df[cols] = scaler.transform(df[cols])
    return df, msg
```

### relative_veh_relations.py

```
import pandas as pd

list_of_vehicles = [
    "precedingId",
    "followingId",
    "leftPrecedingId",
    "leftAlongsideId",
    "leftFollowingId",
    "rightPrecedingId",
    "rightFollowingId",
    "rightAlongsideId",
]

list_of_features = ["x", "y", "xVelocity",
   "yVelocity","vehClass"]


def create_frame_id_pairs(df: pd.DataFrame, ids: str) -> list:
    return [(f, i) for f, i in zip(df.frame.values,
       df[ids].values)]
```

```
def create_feature_map(pairs: list, feature: list) -> dict:
    """ Creates mapping (frame, Id) -> feature """
    return dict(zip(pairs, feature))


def frameId_to_feature(frameId: tuple, feature_map: dict):
    """
    Maps tuple (frame, id) which represents specific vehicle in
       specific
    time to other feature from that frame

    If id is 0 this function returnes 0

    Else returns:
        df.iloc[ index in which frame and id are, feature]
    """
    if frameId[1] == 0:
        return 0
    # elif feature_map[frameId] == 0:
    #     return 0
    else:
        return feature_map[frameId]


def add_feature(df, id_name: str, feature_name: str) ->
   pd.DataFrame:
    frameEgoPair = create_frame_id_pairs(df, "id")
    frameIdPair = create_frame_id_pairs(df, id_name)

    frameEgoFeatureMap = create_feature_map(frameEgoPair,
       df[feature_name].values)

    feature = [
        frameId_to_feature(frameId, frameEgoFeatureMap) for
           frameId in frameIdPair
    ]
    name = id_name[:-2] + feature_name[0].upper() +
       feature_name[1:]
    df[name] = feature
    msg = "Add {} column to dataframe".format(name)
```

```
    return df, msg

def add_veh_relations(df):
    msg = []
    for veh in list_of_vehicles:
        for feature in list_of_features:
            df, m = add_feature(df, veh, feature)
            msg.append(m)
    return df, msg
```

### reporting*features.py*

```
from pylatex import Document, Section, Itemize, Enumerate,
   Description, Command, NoEscape


def create_latex_report(msgs, save_path):
    doc = Document()

    with doc.create(Section('Data processing')):
        with doc.create(Itemize()) as itemize:
            for msg in msgs:
                itemize.add_item(msg)

    doc.generate_pdf(save_path, clean_tex=False)
```

### symetry.py

```
import pandas as pd
import numpy as np

# features to convert
FEATURES =
   ["xVelocity","xAcceleration","yVelocity","yAcceleration"]
# uncomment below when testing
# FEATURES = ["test"]

def get_feature_lane_pairs(df: pd.DataFrame, feature: str) ->
   list:
    return [(f, i) for f, i in zip(df[feature].values,
       df.laneId.values)]
```

```
def convert(feature_lane_pair:tuple) -> float:
    feature, lane = feature_lane_pair
    if lane == 2 or lane == 3:
        feature *= -1
    return feature

def add_symetry(df:pd.DataFrame) -> tuple:
    """Convert values in upper lanes to local coordinate system

    Arguments:
        df {pd.DataFrame} -- [description]

    Returns:
        tuple -- (df, msg)
    """
    for feature in FEATURES:
        feature_lane_pair = get_feature_lane_pairs(df,feature)
        converted_features = list(map(convert,feature_lane_pair))
        df[feature] = converted_features
    msg = ["Converted folowing values to local coordinate
       system: {}".format(FEATURES)]
    return df, msg
```

### symetry_test.py

```
import unittest
import pandas as pd
import symetry as sm


class TestBalancingData(unittest.TestCase):
    def test_convert(self):
        self.assertEqual(sm.convert((-1, 2)), 1)
        self.assertEqual(sm.convert((1, 5)), 1)

    def test_add_symetry(self):
        df = pd.DataFrame({"test": [-1, -1, -1, -1], "laneId":
           [2, 2, 5, 5]})
        df_sym = pd.DataFrame({"test": [1, 1, -1, -1], "laneId":
```

```
            [2, 2, 5, 5]})
        self.assertListEqual(
            sm.add_symetry(df)[0].test.values.tolist(),
                df_sym.test.values.tolist()
        )


if __name__ == "__main__":
    unittest.main()
```

### test_data.py

```
import pandas as pd

df = pd.read_csv("../data/interim/test_data.csv")

df["dhw"] = df.dhw.map(lambda x: 100 if x==0 else x)

df.to_csv("../data/interim/test_data_final.csv")
print("DHW solved")
```

### test_data.py

```
import pandas as pd

df = pd.read_csv("../data/interim/test_data.csv")

df["dhw"] = df.dhw.map(lambda x: 100 if x==0 else x)

df.to_csv("../data/interim/test_data_final.csv")
print("DHW solved")
```

### vehicle_class.py

```
import pandas as pd
import numpy as np

categories = {
    "Truck":0,
    "Car":1
}
```

```
def get_id_class(meta) -> (list,list):
    return meta["id"], meta["class"]

def veh_class_categorize(veh_classes: pd.Series) -> list:
    return veh_classes.map(categories).values

def create_id_class(ids:list,veh_class:np.array) -> dict:
    return dict(zip(ids, veh_class))

def map_classes_to_ids(ids:pd.Series,mapping:dict) -> pd.Series:
    """Creates collumn of vehicle classes from ids in
       dataframe"""
    return  ids.map(mapping)

def input_class(tracks,classes: pd.Series) -> pd.DataFrame:
    """ Add vehClass column to dataframe """
    tracks["vehClass"] = classes
    return tracks


def add_class(tracks,meta):
    msg = "Add vehClass column to dataframe"
    ids,veh_classes = get_id_class(meta)
    veh_classes = veh_class_categorize(veh_classes)
    id_to_class = create_id_class(ids,veh_classes)
    tracks_ids = tracks.id
    final_classes = map_classes_to_ids(tracks_ids,id_to_class)
    return input_class(tracks,final_classes), [msg]
```

### calculate_metrics.py

```
import argparse

import sklearn as sk
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from sklearn.metrics import (
    confusion_matrix,
```

```
    classification_report,
    brier_score_loss,
    roc_curve,
    precision_recall_curve,
    auc
)

parser = argparse.ArgumentParser(description="show metrics")
parser.add_argument("--probs", type=str, default =
   "evaluation/predictions/probs_whole_test_set_model4v2_normal.csv")
args = parser.parse_args()

df = pd.read_csv("../data/interim/test_data_final.csv")
df=df.iloc[:30000,:]
labels = df.laneChangeT100.map(bool)

probs = pd.read_csv(args.probs).probs
preds = probs.map(lambda x: x > 0.6)

print("data_loaded")
cm = confusion_matrix(labels, preds)
print(cm)

cp = classification_report(labels, preds, output_dict=False)
cp_dict = classification_report(labels, preds, output_dict=True)

print(cp)
precision = cp_dict["True"]["precision"]
recall = cp_dict["True"]["recall"]

bs = brier_score_loss(labels, probs)
print(bs)

all_false = [False for _ in range(len(labels))]
bs_avg = brier_score_loss(all_false, probs)
print("Referent BrierScore: ", bs_avg)

bss = 1 - (bs / (bs + bs_avg))
print("BrierSkillScore:", bss)
```

```
fpr, tpr, tresh = roc_curve(labels,probs)
pr, re, tr = precision_recall_curve(labels,probs)

print("PR AUC: ", auc(re,pr))
no_skill = len(labels[labels==1]) / len(labels)
plt.subplot(2,1,1)
plt.scatter(re,pr, c=np.append(tr,1))
plt.plot([0,1],[no_skill,no_skill], ls="--")
plt.xlim([0,1])
plt.scatter(recall, precision, marker="x")
plt.subplot(2,1,2)
plt.plot(tr)
plt.plot([0,len(tr)],[0.3,0.3])

plt.show()
```

### marginals4.py

```
import argparse
import random
import pyro
import torch
import numpy as np
import pandas as pd
from pyro.infer import TraceEnum_ELBO
from pyro import poutine
import matplotlib.pyplot as plt

import trained_model


def get_veh(df, is_random=True, idx=None):
    if is_random:
        veh_id = random.choice(df.sceneId.values)
    else:
        veh_id = "24_{}".format(idx)
    return df[df["sceneId"] == veh_id], veh_id


def keep_change_div(random_veh, frames=100):
```

```
    random_veh_keep =
       random_veh[random_veh["laneChangeT{}".format(frames)] == 0]
    random_veh_change =
       random_veh[random_veh["laneChangeT{}".format(frames)] == 1]
    return random_veh_keep, random_veh_change


def predict_loop(model, input_data: list, observable_features):
    probs = []
    count = 0
    for row in input_data:
        row = torch.tensor(row)
        count += 1
        if count % 1000 == 0:
            print(count, "/", len(input_data))

        data_dict = get_observation_data(
            observable_features, list(map(torch.tensor, row))
        )

        if args.model == "model5":
            data_dict["ax_pvx"] = row[[-3,-1]]
            data_dict["vx_pvx"] = row[[-2,-1]]

        conditioned_model = pyro.condition(model, data=data_dict)

        marginal_intention = TraceEnum_ELBO().compute_marginals(
            conditioned_model, guide, 5
        )["lane_change"]

        probs.append(marginal_intention.probs.item())

    return probs


def get_random_input(df, features: list, lc=False):
    has_lane_change = 0.5
    if lc == True:
        while has_lane_change < 1:
            vehicle_df, idx = get_veh(df, True)
```

```
            has_lane_change =
               np.sum(vehicle_df.laneChange.values)
    else:
        while has_lane_change != 0:
            vehicle_df, idx = get_veh(df, True)
            has_lane_change =
               np.sum(vehicle_df.laneChange.values)
    data = vehicle_df.loc[:, features].values.tolist()
    return data, idx


def get_input(df, features: list, idx):
    vehicle_df, _ = get_veh(df, False, idx)
    data = vehicle_df.loc[:, features].values.tolist()
    return data, idx


def get_observation_data(features, row):
    return dict(zip(features, row))


def save_probs(probs, idx):
    df_probs = pd.DataFrame({"probs": probs})
    df_probs.to_csv("evaluation/predictions/probs_{}.csv".format(idx))


def plot_results(probs, vehicle_df):
    keep, change_1s = keep_change_div(df[df["sceneId"] == idx],
       frames=25)
    _, change_2s = keep_change_div(df[df["sceneId"] == idx],
       frames=50)
    _, change_3s = keep_change_div(df[df["sceneId"] == idx],
       frames=75)
    _, change_4s = keep_change_div(df[df["sceneId"] == idx],
       frames=100)

    u1, u2, u3, l1, l2, l3 = 7.82, 11.91, 15.75, 20.43, 24.39,
       28.36
    plt.subplot(2, 1, 1)
```

```
    plt.scatter(keep.x, keep.y, color="blue")
    plt.scatter(change_4s.x, change_4s.y, color="orange")
    plt.scatter(change_3s.x, change_3s.y, color="red")
    plt.scatter(change_2s.x, change_2s.y, color="orange")
    plt.scatter(change_1s.x, change_1s.y, color="red")

    plt.plot([0, 410], [u1, u1], color="black")
    plt.plot([0, 410], [u2, u2], color="black", ls="--")
    plt.plot([0, 410], [u3, u3], color="black")
    plt.plot([0, 410], [l1, l1], color="black")
    plt.plot([0, 410], [l2, l2], color="black", ls="--")
    plt.plot([0, 410], [l3, l3], color="black")

    plt.grid(True, axis="x")
    plt.ylim([30, 0])
    plt.xlim([0, 410])
    plt.xticks(list(range(0, 410, 25)))
    plt.subplot(2, 1, 2)
    if keep.y.values[0] > 15:
        plt.scatter(vehicle_df.x, probs, color="green", s=2)
    else:
        plt.scatter(vehicle_df.x, probs, color="red", s=2)

    plt.grid(True, axis="x")
    plt.ylim([0, 1.1])
    plt.xlim([0, 410])
    plt.xticks(list(range(0, 410, 25)))
    plt.show()


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Evaluate the
       model")
    parser.add_argument(
        "--random", type=bool, default=True, help="Draw random
           vehicle",
    )
    parser.add_argument(
        "--id", type=int, default=1054, help="Vehicle id,
           overriden if random == True"
```

```
)
parser.add_argument(
    "--model", type=str, default="model5", help="Model you
       want to evaluate"
)
parser.add_argument(
    "--whole_set", type=bool, default=False, help="Predict
       for whole dataset"
)
parser.add_argument(
    "--save_probs", type=bool, default=False, help="Save
       predictions as .csv"
)
parser.add_argument(
    "--plot_results",
    type=bool,
    default=True,
    help="Visualise trajectory and probabilities",
)
parser.add_argument("--lc", type=bool, default=False,
   help="True if you want vehicle with lane change")
args = parser.parse_args()

models = {
    "model3": trained_model.Model3,
    "model4": trained_model.Model4,
    "model4v2": trained_model.Model4v2,
    "model5": trained_model.Model5,
}
observable_features = {
    "model3": ["laneIdRL", "yAcceleration", "yVelocity"],
    "model4": [
        "laneIdRL",
        "yAcceleration",
        "yVelocity",
        "vehClass",
        "precedingVehClass",
    ],
    "model4v2": [
        "laneIdRL",
```

```
            "yAcceleration",
            "yVelocity",
            "vehClass",
            "precedingVehClass",
            "dhw"
        ],
        "model5": [
            "laneIdRL",
            "yAcceleration",
            "yVelocity",
            "vehClass",
            "precedingVehClass",
            "xAcceleration",
            "xVelocity",
            "precedingXVelocity",
        ],
    }
    observable_variables = {
        "model3": ["lane", "a", "v"],
        "model4": ["lane", "a", "v", "ego_class",
           "preceding_class",],
        "model4v2": ["lane", "a", "v", "ego_class",
           "preceding_class","dhw"],
        "model5": ["lane", "a", "v", "ego_class",
           "preceding_class","ax_pvx","vx_pvx"],
    }

    BN = models[args.model]()
    model = BN.model
    guide = BN.guide

    df = pd.read_csv("../data/interim/test_data_final.csv")

    if args.whole_set:
        data = df.loc[:,
           observable_features[args.model]].values.tolist()
    elif args.random:
        data, idx = get_random_input(df,
           observable_features[args.model], lc=args.lc)
    else:
```

```
        data, idx = get_input(df,
            observable_features[args.model], args.id)

    probs = predict_loop(model, data,
        observable_variables[args.model])

    if args.whole_set:
        save_probs(probs,
            "whole_test_set_{}_normal".format(args.model))
    elif args.save_probs:
        save_probs(probs, idx)

    if args.plot_results:
        vehicle_df = df[df["sceneId"] == idx]
        print(vehicle_df.sceneId.iloc[0])
        plot_results(probs, vehicle_df)
```

### trained_model.py

```
import torch
import pyro
import random
import torch.distributions.constraints as constraints
from pyro import distributions as dist
from pyro.contrib.autoguide import AutoGuide, AutoContinuous
from pyro.infer import config_enumerate


def positive_definite(tensor):
    constrained_tensor = torch.zeros_like(tensor)
    if len(tensor.shape) > 2:
        for dim in [0,1]:
            tensor_part = tensor[dim]
            constrained_tensor[dim] =
                torch.add(torch.mm(tensor_part, tensor_part.t()),
                1e-3 * torch.eye(2))
    else:
        constrained_tensor = torch.add(torch.mm(tensor,
            tensor.t()), 0.1 * torch.eye(2))
    return constrained_tensor
```

```
class Model3:
    @config_enumerate
    def model(self, data):
        a = pyro.param("alpha", torch.tensor(10.0),
            constraint=constraints.positive)
        b = pyro.param("beta", torch.tensor(10.0),
            constraint=constraints.positive)
        # with pyro.plate("data", data.shape[0], dim=-1):
        intention = pyro.sample("intention", dist.Beta(a, b))
        a_loc = torch.tensor([[0.0328, -0.0419], [0.1865,
            -0.1461]])
        a_scale = torch.tensor([[0.0940, 0.0986], [0.1466,
            0.1422]])
        v_loc = torch.tensor([[-0.0706, 0.0845], [0.4837,
            -0.4639]])
        v_scale = torch.tensor([[0.2028, 0.2139], [0.3916,
            0.3767]])

        lane_change = pyro.sample("lane_change",
            dist.Bernoulli(intention))

        lane = pyro.sample("lane", dist.Bernoulli(0.5))

        parents = [lane_change.long(), lane.long()]

        pyro.sample("a", dist.Normal(a_loc[parents],
            a_scale[parents]))
        pyro.sample("v", dist.Normal(v_loc[parents],
            v_scale[parents]))

    def guide(self, data):
        return AutoGuide(self.model)


class Model4:
    # @config_enumerate
    def model(self, data):
        ego_class_loc = pyro.param("ego_loc", torch.tensor(0.5))
```

```
preceding_class_loc = pyro.param("prec_loc",
   torch.tensor(0.5))

intention = pyro.param(
    "intention_cpd",
    torch.tensor([[0.04, 0.06], [0.2, 0.15]]),
    constraint=constraints.unit_interval,
)
a_loc = pyro.param("a_loc_p", torch.tensor([[0.0, 0.0],
   [0.2, -0.2]]))
a_scale = pyro.param(
    "a_scale_p",
    torch.tensor([[0.1, 0.1], [0.25, 0.25]]),
    constraint=constraints.positive,
)
v_loc = pyro.param("v_loc_p", torch.tensor([[0.0, 0.0],
   [0.75, -0.75]]))
v_scale = pyro.param(
    "v_scale_p",
    torch.tensor([[0.1, 0.1], [0.25, 0.25]]),
    constraint=constraints.positive,
)
pyro.get_param_store().load("models/saved_models/T100_4_200_0.01_sym

ego_class = pyro.sample("ego_class",
   dist.Bernoulli(ego_class_loc))
preceding_class = pyro.sample(
    "preceding_class",
       dist.Bernoulli(preceding_class_loc)
)

pa_intention = [ego_class.long(), preceding_class.long()]
lane_change = pyro.sample(
    "lane_change",
       dist.Bernoulli(intention[pa_intention]),
    infer={"config_enumerate":"parallel"},
)

lane = pyro.sample("lane", dist.Bernoulli(0.5))
```

```
        parents = [lane_change.long(), lane.long()]

        pyro.sample("a", dist.Normal(a_loc[parents],
            a_scale[parents]))
        pyro.sample("v", dist.Normal(v_loc[parents],
            v_scale[parents]))

    def guide(self, data):
        return AutoGuide(self.model)

class Model4v2:
    @config_enumerate
    def model(self, data):
        ego_class_loc = pyro.param("ego_loc", torch.tensor(0.5))
        preceding_class_loc = pyro.param("prec_loc",
            torch.tensor(0.5))

        intention = pyro.param(
            "intention_cpd",
            torch.tensor([[0.04, 0.06], [0.2, 0.15]]),
            constraint=constraints.unit_interval,
        )
        a_loc = pyro.param("a_loc_p", torch.tensor([[0.0, 0.0],
            [0.2, -0.2]]))
        a_scale = pyro.param(
            "a_scale_p",
            torch.tensor([[0.1, 0.1], [0.25, 0.25]]),
            constraint=constraints.positive,
        )
        v_loc = pyro.param("v_loc_p", torch.tensor([[0.0, 0.0],
            [0.75, -0.75]]))
        v_scale = pyro.param(
            "v_scale_p",
            torch.tensor([[0.1, 0.1], [0.25, 0.25]]),
            constraint=constraints.positive,
        )
        dhw_param_a = pyro.param(
            "dhw_param_a",
            torch.tensor([[3, 3], [3, 3]], dtype=float),
            constraint=constraints.positive,
```

```
        )

        pyro.get_param_store().load("models/saved_models/T100_4v2_100_0.3_no

        ego_class = pyro.sample("ego_class",
           dist.Bernoulli(ego_class_loc))
        preceding_class = pyro.sample(
            "preceding_class",
               dist.Bernoulli(preceding_class_loc)
        )

        pa_intention = [ego_class.long(), preceding_class.long()]
        lane_change = pyro.sample(
            "lane_change",
               dist.Bernoulli(intention[pa_intention]),
            infer={"config_enumerate":"parallel"},
        )

        lane = pyro.sample("lane", dist.Bernoulli(0.5))

        parents = [lane_change.long(), lane.long()]
        pyro.sample("dhw", dist.HalfNormal(dhw_param_a[parents]))
        pyro.sample("a", dist.Normal(a_loc[parents],
           a_scale[parents]))
        pyro.sample("v", dist.Normal(v_loc[parents],
           v_scale[parents]))

    def guide(self, data):
        return AutoGuide(self.model)

class Model5:
    @config_enumerate
    def model(self, data):
        ego_class_loc = pyro.param(
            "ego_loc", torch.tensor(0.6),
               constraint=constraints.unit_interval
        )
        preceding_class_loc = pyro.param(
            "prec_loc", torch.tensor(0.6),
               constraint=constraints.unit_interval
```

```
)
intention = pyro.param(
    "intention_cpd",
    torch.tensor([[0.04, 0.06], [0.2, 0.15]]),
    constraint=constraints.unit_interval,
)
a_loc = pyro.param("a_loc_p", torch.tensor([[0.0, 0.0],
   [0.2, -0.2]]))
a_scale = pyro.param(
    "a_scale_p",
    torch.tensor([[0.1, 0.1], [0.25, 0.25]]),
    constraint=constraints.positive,
)
v_loc = pyro.param("v_loc_p", torch.tensor([[0.0, 0.0],
   [0.75, -0.75]]))
v_scale = pyro.param(
    "v_scale_p",
    torch.tensor([[0.1, 0.1], [0.25, 0.25]]),
    constraint=constraints.positive,
)

vx_loc = pyro.param("vx_loc_p", torch.randn(2, 2, 2, 2))
ax_loc = pyro.param("ax_loc_p", torch.randn(2, 2, 2, 2))
vx_scale = pyro.param(
    "vx_scale",
    torch.randn(2, 2, 2, 2, 2).float(),
    # constraint=constraints.positive_definite,
)
ax_scale = pyro.param(
    "ax_scale",
    torch.randn(2, 2, 2, 2, 2).float(),
    # constraint=constraints.positive_definite,
)

pyro.get_param_store().load("models/saved_models/T100_5_200_0.02_sym

ego_class = pyro.sample("ego_class",
   dist.Bernoulli(ego_class_loc),)
preceding_class = pyro.sample(
    "preceding_class",
```

```
        dist.Bernoulli(preceding_class_loc),
    )
    pa_intention = [ego_class.long(), preceding_class.long()]
    lane_change = pyro.sample(
        "lane_change",
        dist.Bernoulli(intention[pa_intention]),
    )

    lane = pyro.sample("lane", dist.Bernoulli(0.5),)

    parents_avy = [lane_change.long(), lane.long()]

    pyro.sample(
        "ay", dist.Normal(a_loc[parents_avy],
        a_scale[parents_avy]),
    )
    pyro.sample(
        "vy", dist.Normal(v_loc[parents_avy],
        v_scale[parents_avy]),
    )

    parents_avx =
       [ego_class.long(),lane.long(),lane_change.long()]

    ax_covariance =
       positive_definite(ax_scale[parents_avx]).float()
    vx_covariance =
       positive_definite(vx_scale[parents_avx]).float()

    pyro.sample(
        "ax_pvx",
        dist.MultivariateNormal(ax_loc[parents_avx],
        ax_covariance),
    )
    pyro.sample(
        "vx_pvx",
        dist.MultivariateNormal(vx_loc[parents_avx],
        vx_covariance),
    )
```

```
    def guide(self, data):
        return AutoGuide(self.model)
```