A picture containing graphical user interface

Description automatically generated

**Inputs and Outputs to Prediction**

A prediction module uses a map and data from sensor fusion to generate predictions for what all other **dynamic** objects in view are likely to do. To make this clearer, let's look at an example (in json format) of what the **input** to and **output** from prediction might look like.

**Example Input - Sensor Fusion**

{

"timestamp" : 34512.21,

"vehicles" : [

{

"id" : 0,

"x" : -10.0,

"y" : 8.1,

"v\_x" : 8.0,

"v\_y" : 0.0,

"sigma\_x" : 0.031,

"sigma\_y" : 0.040,

"sigma\_v\_x" : 0.12,

"sigma\_v\_y" : 0.03,

},

{

"id" : 1,

"x" : 10.0,

"y" : 12.1,

"v\_x" : -8.0,

"v\_y" : 0.0,

"sigma\_x" : 0.031,

"sigma\_y" : 0.040,

"sigma\_v\_x" : 0.12,

"sigma\_v\_y" : 0.03,

},

]

}

**Example Output**

{

"timestamp" : 34512.21,

"vehicles" : [

{

"id" : 0,

"length": 3.4,

"width" : 1.5,

"predictions" : [

{

"probability" : 0.781,

"trajectory" : [

{

"x": -10.0,

"y": 8.1,

"yaw": 0.0,

"timestamp": 34512.71

},

{

"x": -6.0,

"y": 8.1,

"yaw": 0.0,

"timestamp": 34513.21

},

{

"x": -2.0,

"y": 8.1,

"yaw": 0.0,

"timestamp": 34513.71

},

{

"x": 2.0,

"y": 8.1,

"yaw": 0.0,

"timestamp": 34514.21

},

{

"x": 6.0,

"y": 8.1,

"yaw": 0.0,

"timestamp": 34514.71

},

{

"x": 10.0,

"y": 8.1,

"yaw": 0.0,

"timestamp": 34515.21

},

]

},

{

"probability" : 0.219,

"trajectory" : [

{

"x": -10.0,

"y": 8.1,

"yaw": 0.0,

"timestamp": 34512.71

},

{

"x": -7.0,

"y": 7.5,

"yaw": -5.2,

"timestamp": 34513.21

},

{

"x": -4.0,

"y": 6.1,

"yaw": -32.0,

"timestamp": 34513.71

},

{

"x": -3.0,

"y": 4.1,

"yaw": -73.2,

"timestamp": 34514.21

},

{

"x": -2.0,

"y": 1.2,

"yaw": -90.0,

"timestamp": 34514.71

},

{

"x": -2.0,

"y":-2.8,

"yaw": -90.0,

"timestamp": 34515.21

},

]

}

]

},

{

"id" : 1,

"length": 3.4,

"width" : 1.5,

"predictions" : [

{

"probability" : 1.0,

"trajectory" : [

{

"x": 10.0,

"y": 12.1,

"yaw": -180.0,

"timestamp": 34512.71

},

{

"x": 6.0,

"y": 12.1,

"yaw": -180.0,

"timestamp": 34513.21

},

{

"x": 2.0,

"y": 12.1,

"yaw": -180.0,

"timestamp": 34513.71

},

{

"x": -2.0,

"y": 12.1,

"yaw": -180.0,

"timestamp": 34514.21

},

{

"x": -6.0,

"y": 12.1,

"yaw": -180.0,

"timestamp": 34514.71

},

{

"x": -10.0,

"y": 12.1,

"yaw": -180.0,

"timestamp": 34515.21

}

]

}

]

}

]

}

**Notes**

1. The predicted trajectories shown here only extend out a few seconds. In reality the predictions we make extend to a horizon of 10-20 seconds.
2. The trajectories shown have 0.5 second resolution. In reality we would generate slightly finer-grained predictions.
3. This example only shows vehicles but in reality we would also generate predictions for **all** dynamic objects in view.

# Frenet Coordinates

Before we discuss process models, we should mention "Frenet Coordinates", which are a way of representing position on a road in a more intuitive way than traditional (x,y)(*x*,*y*) Cartesian Coordinates.

With Frenet coordinates, we use the variables s*s* and d*d* to describe a vehicle's position on the road. The s*s* coordinate represents distance along the road (also known as longitudinal displacement) and the d*d* coordinate represents side-to-side position on the road (also known as lateral displacement).

Why do we use Frenet coordinates? Imagine a curvy road like the one below with a Cartesian coordinate system laid on top of it...

Diagram

Description automatically generated

Using these Cartesian coordinates, we can try to describe the path a vehicle would normally follow on the road...

A picture containing shape

Description automatically generated

Chart

Description automatically generated

And notice how curvy that path is! If we wanted equations to describe this motion it wouldn't be easy!

x(t) = \text{?}*x*(*t*)=?

y(t) = \text{?}*y*(*t*)=?

Ideally, it should be mathematically easy to describe such common driving behavior. But how do we do that? One way is to use a new coordinate system. Now instead of laying down a normal Cartesian grid, we do something like you see below...

Chart, radar chart

Description automatically generated

Here, we've defined a new system of coordinates. At the bottom we have s=0*s*=0 to represent the beginning of the segment of road we are thinking about and d=0*d*=0 to represent the center line of that road. To the left of the center line we have negative d*d* and to the right d*d* is positive.

So what does a typical trajectory look like when presented in Frenet coordinates?

Chart, radar chart

Description automatically generated

Chart, line chart

Description automatically generated

It looks straight!

In fact, if this vehicle were moving at a constant speed of v\_0*v*0​ we could write a mathematical description of the vehicle's position as:

s(t) = v\_0t*s*(*t*)=*v*0​*t*

d(t) = 0*d*(*t*)=0

We'll be working with Frenet coordinates a good deal in the rest of the course, because...

Chart, scatter chart

Description automatically generated

...straight lines are so much easier than curved ones.

# More on Process Models

Later in the lesson I'm going to ask you to read a paper titled ["A comparative study of multiple-model algorithms for maneuvering target tracking"](https://d17h27t6h515a5.cloudfront.net/topher/2017/June/5953fc34_a-comparative-study-of-multiple-model-algorithms-for-maneuvering-target-tracking/a-comparative-study-of-multiple-model-algorithms-for-maneuvering-target-tracking.pdf) but for now I'd like you to take a look at section 3.1 and 3.2 only. This section, titled MM Tracking Algorithms' Design, discusses the 9 process models used in the earlier part of the paper.

Before you read the section, I'll explain some of the uncommon notation you will see.

### Notes on Notation

#### 1. Matrix Notation

When you see something like the following:F\_{CV} = \text{diag}[F\_2, F\_2], F\_2 = \begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix}*FCV*​=diag[*F*2​,*F*2​],*F*2​=[10​*T*1​]

it means that F*F* is a 4x4 matrix, with F\_{2\_{}}*F*2​​ as blocks along the diagonal. Written out fully, this means:F\_{CV} = \begin{bmatrix} 1 & T & 0 & 0\\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & T \\ 0 & 0 & 0 & 1 \end{bmatrix}*FCV*​=⎣⎢⎢⎡​1000​*T*100​0010​00*T*1​⎦⎥⎥⎤​

#### 2. State Space

The process models all use cartesian coordinates. The state space is\mathbf{x} = \begin{bmatrix} x\\ \dot{x} \\ y \\ \dot{y} \end{bmatrix}**x**=⎣⎢⎢⎡​*xx*˙*yy*˙​​⎦⎥⎥⎤​

#### 3. Variables

The equation x\_{k} = Fx\_{k-1} + Gu\_{k-1} + Gw\_k, \ \ w\_k \sim \mathcal{N}(0,Q)*xk*​=*Fxk*−1​+*Guk*−1​+*Gwk*​,  *wk*​∼N(0,*Q*) should be read as follows:

*the predicted state at time k (x\_{k\_{}}xk​​) is given by evolving (FF) the previous state (x\_{k-1\_{}}xk−*1*​​), incorporating (GG) the controls (u\_{k-1\_{}}uk−*1*​​) given at the previous time step, and adding normally distributed noise (w\_kwk​).*

## The Paper

You can find the paper here: [A comparative study of multiple-model algorithms for maneuvering target tracking](https://d17h27t6h515a5.cloudfront.net/topher/2017/June/5953fc34_a-comparative-study-of-multiple-model-algorithms-for-maneuvering-target-tracking/a-comparative-study-of-multiple-model-algorithms-for-maneuvering-target-tracking.pdf)

**Summary so Far**

So far you have learned about the two main approaches to prediction.

**1. Data-Driven Approaches**

Data-driven approaches solve the prediction problem in two phases:

1. Offline training
2. Online Prediction

**1.1 Offline Training**

In this phase the goal is to feed some machine learning algorithm a lot of data to train it. For the trajectory clustering example this involved:

1. **Define similarity** - we first need a definition of similarity that agrees with human common-sense definition.
2. **Unsupervised clustering** - at this step some machine learning algorithm clusters the trajectories we've observed.
3. **Define Prototype Trajectories** - for each cluster identify some small number of typical "prototype" trajectories.

**1.2 Online Prediction**

Once the algorithm is trained we bring it onto the road. When we encounter a situation for which the trained algorithm is appropriate (returning to an intersection for example) we can use that algorithm to actually predict the trajectory of the vehicle. For the intersection example this meant:

1. **Observe Partial Trajectory** - As the target vehicle drives we can think of it leaving a "partial trajectory" behind it.
2. **Compare to Prototype Trajectories** - We can compare this partial trajectory to the *corresponding parts* of the prototype trajectories. When these partial trajectories are more similar (using the same notion of similarity defined earlier) their likelihoods should increase relative to the other trajectories.
3. **Generate Predictions** - For each cluster we identify the most likely prototype trajectory. We broadcast each of these trajectories along with the associated probability (see the image below).

A picture containing chart

Description automatically generated

**2. Model Based Approaches**

You can think of model based solutions to the prediction problem as also having an "offline" and online component. In that view, this approach requires:

1. *Defining* process models (offline).
2. *Using* process models to compare driver behavior to what would be expected for each model.
3. *Probabilistically classifying* driver intent by comparing the likelihoods of various behaviors with a multiple-model algorithm.
4. *Extrapolating* process models to generate trajectories.

**2.1 Defining Process Models**

You saw how process models can vary in complexity from very simple...\large \begin{bmatrix} \dot{s}\\ \dot{d} \end{bmatrix} = \begin{bmatrix} s\_{0} \\ 0 \end{bmatrix} + \mathbf{w}[*s*˙*d*˙​]=[*s*0​0​]+**w**

to very complex...

\large \begin{bmatrix} \ddot{s} \\ \ddot{d} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} \dot{\theta}\dot{d} + a\_s \\ -\dot{\theta}\dot{s} + \frac{2}{m}(F\_{c,f}\cos\delta + F\_{c,r}) \\ \frac{2}{I\_z} (l\_f F\_{c,f} - l\_rF\_{c,r}) \end{bmatrix} + \mathbf{w}⎣⎢⎡​*s*¨*d*¨*θ*¨​⎦⎥⎤​=⎣⎢⎡​*θ*˙*d*˙+*as*​−*θ*˙*s*˙+*m*2​(*Fc*,*f*​cos*δ*+*Fc*,*r*​)*Iz*​2​(*lf*​*Fc*,*f*​−*lr*​*Fc*,*r*​)​⎦⎥⎤​+**w**

**2.2 Using Process Models**

Process Models are first used to compare a target vehicle's observed behavior to the behavior we would expect for each of the maneuvers we've created models for. The pictures below help explain how process models are used to calculate these likelihoods.

Diagram

Description automatically generated

On the left we see two images of a car. At time k-1*k*−1 we predicted where the car would be if it were to go straight vs go right. Then at time k*k* we look at where the car actually is. The graph on the right shows the car's observed s*s* coordinate along with the probability distributions for where we *expected* the car to be at that time. In this case, the s*s* that we observe is substantially more consistent with turning right than going straight.

**2.3 Classifying Intent with Multiple Model Algorithm**

In the image at the top of the page you can see a bar chart representing probabilities of various *clusters* over time. Multiple model algorithms serve a similar purpose for model based approaches: they are responsible for maintaining beliefs for the probability of each maneuver. The algorithm we discussed is called the **Autonomous Multiple Model** algorithm (AMM). AMM can be summarized with this equation:

\large \mu\_k^{(i)} = \frac{\mu\_{k-1}^{(i)}L\_k^{(i)}}{\sum\_{j=1}^M\mu\_{k-1}^{(j)}L\_k^{(j)}}*μk*(*i*)​=∑*j*=1*M*​*μk*−1(*j*)​*Lk*(*j*)​*μk*−1(*i*)​*Lk*(*i*)​​

or, if we ignore the denominator (since it just serves to normalize the probabilities), we can capture the essence of this algorithm with

\mu\_k^{(i)} \propto \mu\_{k-1}^{(i)}L\_k^{(i)}*μk*(*i*)​∝*μk*−1(*i*)​*Lk*(*i*)​

where the \mu\_k^{(i)}*μk*(*i*)​ is the probability that model number i*i* is the correct model at time k*k* and L\_k^{(i)}*Lk*(*i*)​ is the **likelihood** for that model (as computed by comparison to process model).

The paper, ["A comparative study of multiple model algorithms for maneuvering target tracking"](https://d17h27t6h515a5.cloudfront.net/topher/2017/June/5953fc34_a-comparative-study-of-multiple-model-algorithms-for-maneuvering-target-tracking/a-comparative-study-of-multiple-model-algorithms-for-maneuvering-target-tracking.pdf) is a good reference to learn more.

**2.4 Trajectory Generation**

Trajectory generation is straightforward once we have a process model. We simply iterate our model over and over until we've generated a prediction that spans whatever time horizon we are supposed to cover. Note that each iteration of the process model will necessarily add uncertainty to our prediction.

**Implementing Naive Bayes**

In this exercise you will implement a Gaussian Naive Bayes classifier to predict the behavior of vehicles on a highway. In the image below you can see the behaviors you'll be looking for on a 3 lane highway (with lanes of 4 meter width). The dots represent the d (y axis) and s (x axis) coordinates of vehicles as they either...

1. change lanes left (shown in blue)
2. keep lane (shown in black)
3. or change lanes right (shown in red)

Diagram

Description automatically generated

Your job is to write a classifier that can predict which of these three maneuvers a vehicle is engaged in given a single coordinate (sampled from the trajectories shown below).

Each coordinate contains 4 features:

* s*s*
* d*d*
* \dot{s}*s*˙
* \dot{d}*d*˙

You also know the **lane width** is 4 meters (this might be helpful in engineering additional features for your algorithm).

**Instructions**

1. Implement the train(data, labels) method in the class GNB in classifier.cpp.

Training a Gaussian Naive Bayes classifier consists of computing and storing the mean and standard deviation from the data for each label/feature pair. For example, given the label "change lanes left” and the feature \dot{s}*s*˙, it would be necessary to compute and store the mean and standard deviation of \dot{s}*s*˙ over all data points with the "change lanes left” label.

Additionally, it will be convenient in this step to compute and store the prior probability p(C\_k) for each label C\_k. This can be done by keeping track of the number of times each label appears in the training data.

1. Implement the predict(observation) method in classifier.cpp.

Given a new data point, prediction requires two steps:

* 1. **Compute the conditional probabilities for each feature/label combination**. For a feature x*x* and label C*C* with mean \mu*μ* and standard deviation \sigma*σ* (computed in training), the conditional probability can be computed using the formula [here](https://en.wikipedia.org/wiki/Naive_Bayes_classifier#Gaussian_naive_Bayes):

Here v*v* is the value of feature x*x* in the new data point.

* 1. **Use the conditional probabilities in a Naive Bayes classifier.** This can be done using the formula [here](https://en.wikipedia.org/wiki/Naive_Bayes_classifier#Constructing_a_classifier_from_the_probability_model):

*y*=*k*∈(1,…,*K*)*argmax*​*p*(*Ck*​)*i*=1∏*n*​*p*(*xi*​=*vi*​∣*Ck*​)

In this formula, the argmax is taken over all possible labels C\_k*Ck*​ and the product is taken over all features x\_i*xi*​ with values v\_i*vi*​.

1. When you want to test your classifier, run Test Run and check out the results.

**NOTE**: You are welcome to use some existing implementation of a Gaussian Naive Bayes classifier. But to get the **best** results you will still need to put some thought into what **features** you provide the algorithm when classifying. Though you will only be given the 4 coordinates listed above, you may find that by "engineering" features you may get better performance. For example: the raw value of the d*d* coordinate may not be that useful. But d % lane\_width might be helpful since it gives the *relative* position of a vehicle in it's lane regardless of which lane the vehicle is in.

**Helpful Resources**

* [sklearn documentation on GaussianNB](http://scikit-learn.org/stable/modules/naive_bayes.html#gaussian-naive-bayes)
* [Wikipedia article on Naive Bayes / GNB](https://en.wikipedia.org/wiki/Naive_Bayes_classifier#Gaussian_naive_Bayes)

**Extra Practice**

Provided in one of the links below is python\_extra\_practice, which is the same problem but written in Python that you can optionally go through for extra coding practice. The Python solution is available at the python\_solution link. If you get stuck on the quiz see if you can convert the python solution to C++ and pass the classroom quiz with it. The last link Nd013\_Pred\_Data has all the training and testing data for this problem in case you want to run the problem offline.