ENPM 808A

Introduction to Machine Learning

Final Project

By

Jai Sharma

UNIVERSITY OF MARYLAND 1856

Table of Contents

Introduction

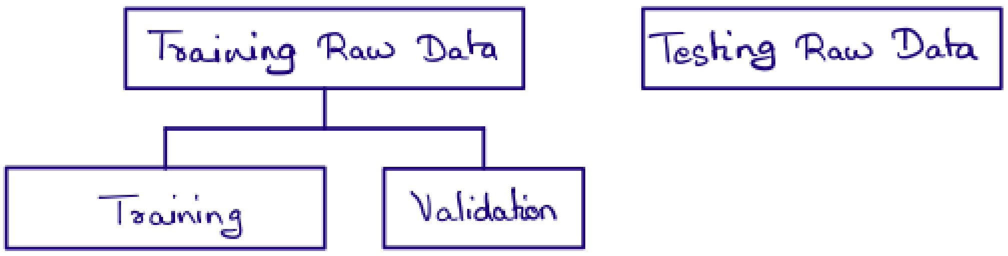
We are provided with a novel dataset on which we need to perform regression. The data comes from multiple path planning MPC tests conducted with a car-like robot model. The data is stored in multiple CSV files and organized in columns as follows:

- *Laser Range - 1:1080*
- *Final Goal (x,y,qk,qr) – 1081:1084*
- *Local Goal (x,y,qk,qr) – 1085:1088*
- *Pose (x,y,qk,qr) – 1089:1090*
- *Cmd_vel_v - 1091*
- *Cmd_vel_w - 1092*

The goal is to build a reliable Machine Learning model that uses the provided data to approximate the command actions translational (v), and angular velocities (w) of the robot.

Data Cleaning and Formatting

To process the tabular data, I use a software library called Pandas. It offers tools to assist with data analysis and data manipulation. I read the csv files and store them in dataframes data type. The general split in data is illustrated in the diagram below:



Training Data

There is an abundance of data available to train on. I import and concatenate data from multiple separate csv files for training. It was important to separate the data in terms of the testing environments. The 2 scenarios I focused on were as follows:

1. A corridor scenario with moving obstacles
2. An open box/hall environment with moving obstacles

Hence a training, validation and testing set were created for each scenario and they were not mixed.

Validation Data

The training set is shuffled and a validation set is extracted from it using sklearn's 'train_test_split' function. All the models I use involve a validation_split parameter of 0.20 i.e. 20%. The validation set is used exclusively for Hyperparameter Tuning. Here is a summary of the training/validation data split for all three sets:

Table 1: Training and Validation Set Variations

Set Type	CSV Files Concatenated	Total Data Size	Training Data Size	Validation Data Size
Box	6	118623	88967	29656
Corridor	5	94780	71085	23695

Testing Data

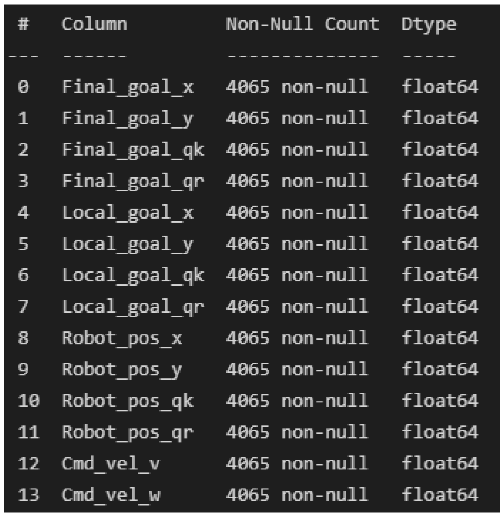
There are several csv files available to test on. I perform tests on roughly 25000 data points for each scenario. These files are used consistently throughout the model study. Here is a summary of the testing sets:

Table 2: Testing Set Variations

Set Type	Set Name	CSV Files Concatenated	Test Data Size
Box	Box_Set	2	26358
Corridor	Corridor_Set	1	24360

Data Imputation

I checked if there are any missing data points or any null-type data points. Empty values can throw errors later in the ML pipeline. To check for this, I used pandas dataframe.info() method. Since all entries are non-null, no statistical imputation techniques were required.



```
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Final_goal_x   4065 non-null   float64
 1   Final_goal_y   4065 non-null   float64
 2   Final_goal_qk  4065 non-null   float64
 3   Final_goal_qr  4065 non-null   float64
 4   Local_goal_x   4065 non-null   float64
 5   Local_goal_y   4065 non-null   float64
 6   Local_goal_qk  4065 non-null   float64
 7   Local_goal_qr  4065 non-null   float64
 8   Robot_pos_x    4065 non-null   float64
 9   Robot_pos_y    4065 non-null   float64
 10  Robot_pos_qk   4065 non-null   float64
 11  Robot_pos_qr   4065 non-null   float64
 12  Cmd_vel_v      4065 non-null   float64
 13  Cmd_vel_w      4065 non-null   float64
```

Image 1: Null-Count for some sample features

Feature Engineering

Feature Reduction

The raw data set offers 1092 features to choose from. It was important to reduce the dimensionality for the following reasons:

- Reduce training time
- Reduce computation time
- Reduce overfitting by encouraging a simpler model

Logic Based Reduction: There are 1080 columns for Lidar data. The resolution is 0.25°. While that can be useful for applications like map building and object detection, it might be too much data for a machine learning problem.

I decided to reduce the lidar data set by averaging every 60 columns. This implies that I used the average lidar reading for 18 regions around the robot, i.e. every 15°.. Having previously worked on Lidar based projects, I feel 15°. should be broad enough to provide the robot with a sense of obstacles presence and empty spaces. This reduced the lidar feature space by 60 times, from 1080 columns to 18 columns.

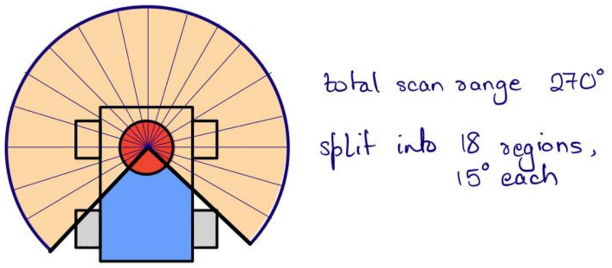


Image 2: Lidar Data split into 18 regions [15°. each]

This leaves the dataset with 18 laser features and 12 additional pose-based features, amounting to a total of 30 features.

Feature Selection

Using domain knowledge of robotics and path planning, it seemed appropriate to eliminate the Final Position data. The plot below shows that there is not much variation in the final goal values for a given file set of data, while the prediction variables v and w are fairly well distributed.

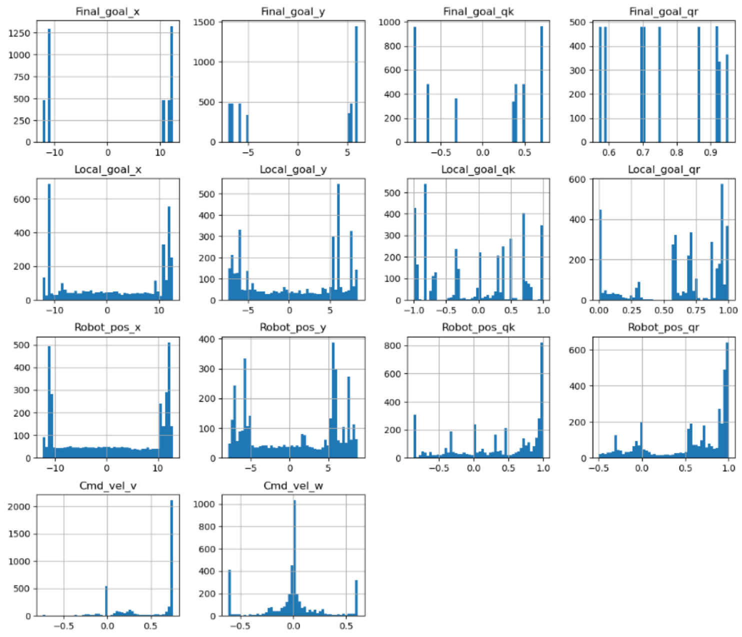


Image 3: Histogram plots for Non-Lidar features

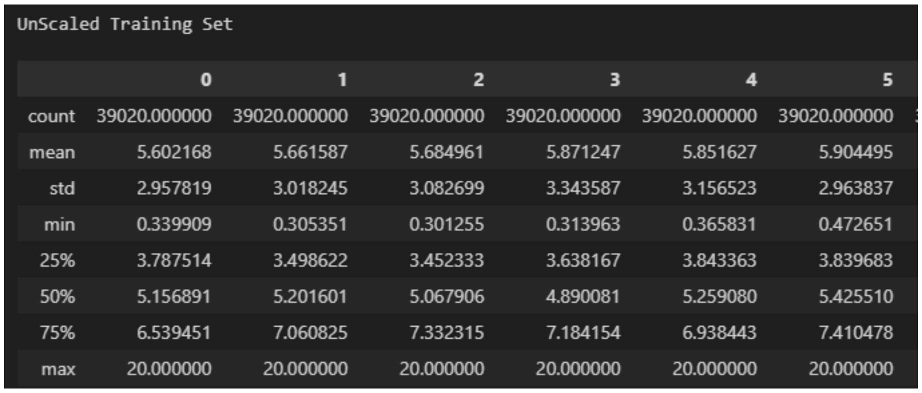
Furthermore, it is highly possible that the local goal in the path planning algorithm could be influenced by the distance between pose and final goal. This could lead to a bigger problem of multicollinearity where an independent feature is influenced by another independent feature. This reduces the total features to 23.

Outliers

It is fair to expect that there are unaccounted factors affecting the data collection process. Hence, it is important to check for and deal with outliers. I used the same plots in Image 3 to assess the presence of outliers. Other than Final Pose, the remaining features are well distributed. For most, the frequency is highest on the limits, so the outliers would be marginally wrong if any. So, I take no further action to deal with outliers.

Feature Scaling

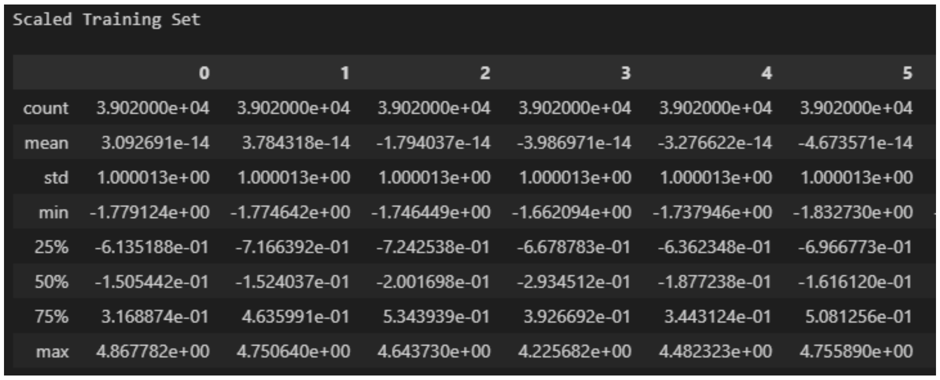
To get the best output from the ML models, I scale the feature data for Training, Validation and Testing. The main idea is to normalize/standardize i.e. $\mu = 0$ and $\sigma = 1$ all the feature columns in X, individually. This assumes that the data is normally distributed. I perform this using sklearn's Standard Scaler function. Following best practices, the data is fitted on the basis of training set and blindly applied to validation and testing set. I do this for each Training Set. Below you can find the statistics change for a sample training file and for sample feature points within it.



UnScaled Training Set

	0	1	2	3	4	5
count	39020.000000	39020.000000	39020.000000	39020.000000	39020.000000	39020.000000
mean	5.602168	5.661587	5.684961	5.871247	5.851627	5.904495
std	2.957819	3.018245	3.082699	3.343587	3.156523	2.963837
min	0.339909	0.305351	0.301255	0.313963	0.365831	0.472651
25%	3.787514	3.498622	3.452333	3.638167	3.843363	3.839683
50%	5.156891	5.201601	5.067906	4.890081	5.259080	5.425510
75%	6.539451	7.060825	7.332315	7.184154	6.938443	7.410478
max	20.000000	20.000000	20.000000	20.000000	20.000000	20.000000

Image 4: shows descriptive statistics for raw data



Scaled Training Set

	0	1	2	3	4	5
count	3.902000e+04	3.902000e+04	3.902000e+04	3.902000e+04	3.902000e+04	3.902000e+04
mean	3.092691e-14	3.784318e-14	-1.794037e-14	-3.986971e-14	-3.276622e-14	-4.673571e-14
std	1.000013e+00	1.000013e+00	1.000013e+00	1.000013e+00	1.000013e+00	1.000013e+00
min	-1.779124e+00	-1.774642e+00	-1.746449e+00	-1.662094e+00	-1.737946e+00	-1.832730e+00
25%	-6.135188e-01	-7.166392e-01	-7.242538e-01	-6.678783e-01	-6.362348e-01	-6.966773e-01
50%	-1.505442e-01	-1.524037e-01	-2.001698e-01	-2.934512e-01	-1.877238e-01	-1.616120e-01
75%	3.168874e-01	4.635991e-01	5.343939e-01	3.926692e-01	3.443124e-01	5.081256e-01
max	4.867782e+00	4.750640e+00	4.643730e+00	4.225682e+00	4.482323e+00	4.755890e+00

Image 5: shows descriptive statistics for scaled data [μ = 0 and σ = 1]

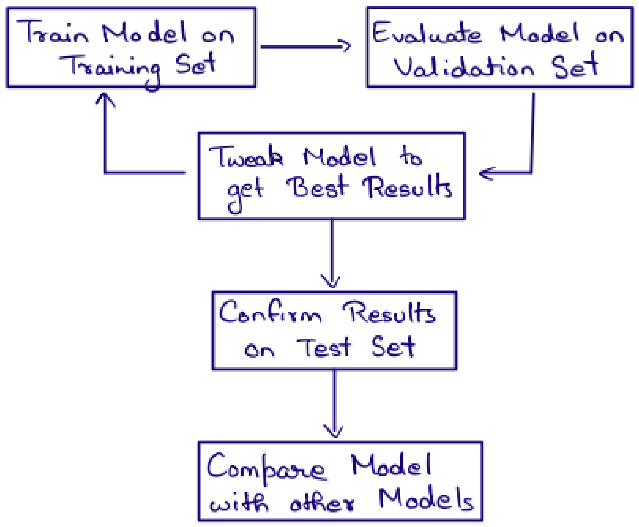
Machine Learning Strategies

The project was a good opportunity to implement novel Machine Learning models and compare the results to what's been taught through the ENPM808A course. Following data cleaning and feature engineering, I started training the chosen Machine Learning models. I used a consistent Pipeline on every model.

The strategy was as follows:

- Train model with default parameters
- Run prediction on Training Data and Testing Data
- Compute R^2 score and Mean Square Error
- Tune Hyperparameters
  - Build parameter dictionary
  - Fit data on Validation Set to find best parameters
- Evaluate best model on Training Data and Testing Data
- Compute R^2 score and Mean Square Error for best version

This strategy is individually applied on all models. The evaluation metrics for the Best version of each model is compared and accordingly the Best Model is chosen. The cyclic nature of training and tuning is illustrated below.


Train Model on Training Set
Evaluate Model on Validation Set
Tweak Model to get Best Results
Confirm Results on Test Set
Compare Model with other Models

It is also important to mention that the models were trained to learn a model for v and w separately in most cases. Models like SGD, XG Boost and Random Forest are not great with learning multioutput regressions. In contrast, it is much simpler to build multioutput prediction models using a neural network. Hence, the models I used, trained and tested for v and w separately.

Evaluation metrics

The first metric used to evaluate was Mean Squared Error [MSE].

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \tilde{y}_i)^2$$

The trained model is run on a Training and Test set. The difference between the predicted and true value is required to compute the metric. It acts as a training heuristic and judges the model's accuracy. Small MSE value signifies a better model.

I also use a second metric called R-squared or R^2 or coefficient of determination. I found it to be a better metric to use for model performance comparison and model selection.

$$R^2 = 1 - \frac{SSE}{SST}$$

SSE is the residual sum of squares and SST is the total sum of squares. The R^2 values vary between 0 and 1. R^2 = 1 is the best case and R^2 = 0 is the worst case.

Sklearn.metric module can compute these prediction errors through functions like mean_squared_error and r2_score.

Hyperparameter Tuning

Parameter and hyperparameters are tuned to find the optimal values that maximize the performance of the model architecture. For each model, relevant parameters are selected, and a range of values are stored in a dictionary. To find this parameter, I implement a randomized search approach, using a sklearn function called RandomizedSearchCV.

RandomizedSearchCV implements a “fit” and a “score” method. In contrast to GridSearchCV, not all parameter values are tried out, but rather a fixed number of parameter settings is sampled from the specified distributions. To compare Grid Search with Randomized Search, I ran the tuning algorithm on Decision Tree Regression Model. Running permutations for params dictionary, Grid Search took 2 minute 6 seconds while Randomized Search only required 3.1 seconds. While it found the best results, it was at a much higher computational cost. Given the high spaces of parameter search, I chose to use Randomized Search.

The initial range of parameters were picked around the default parameters values. If the best params are on the extremes of the provided range, I would expand or slide the range to explore more promising parameters.

Learning Curve

I plotted a learning curve to assess how the model accuracy responds to a change in training data size. This was conducted for a basic SGD Linear regression model, XG Boost model and. As can been seen in the plots below, accuracy converges after 50000 data points. An increase in size of course is expected to help but it looks likes the benefit to accuracy is minimal. Furthermore, the computation cost and training time is something to consider when choosing to increase the training size. Learning from this plot, a training data size of 70,000 points should be acceptable for this learning problem.

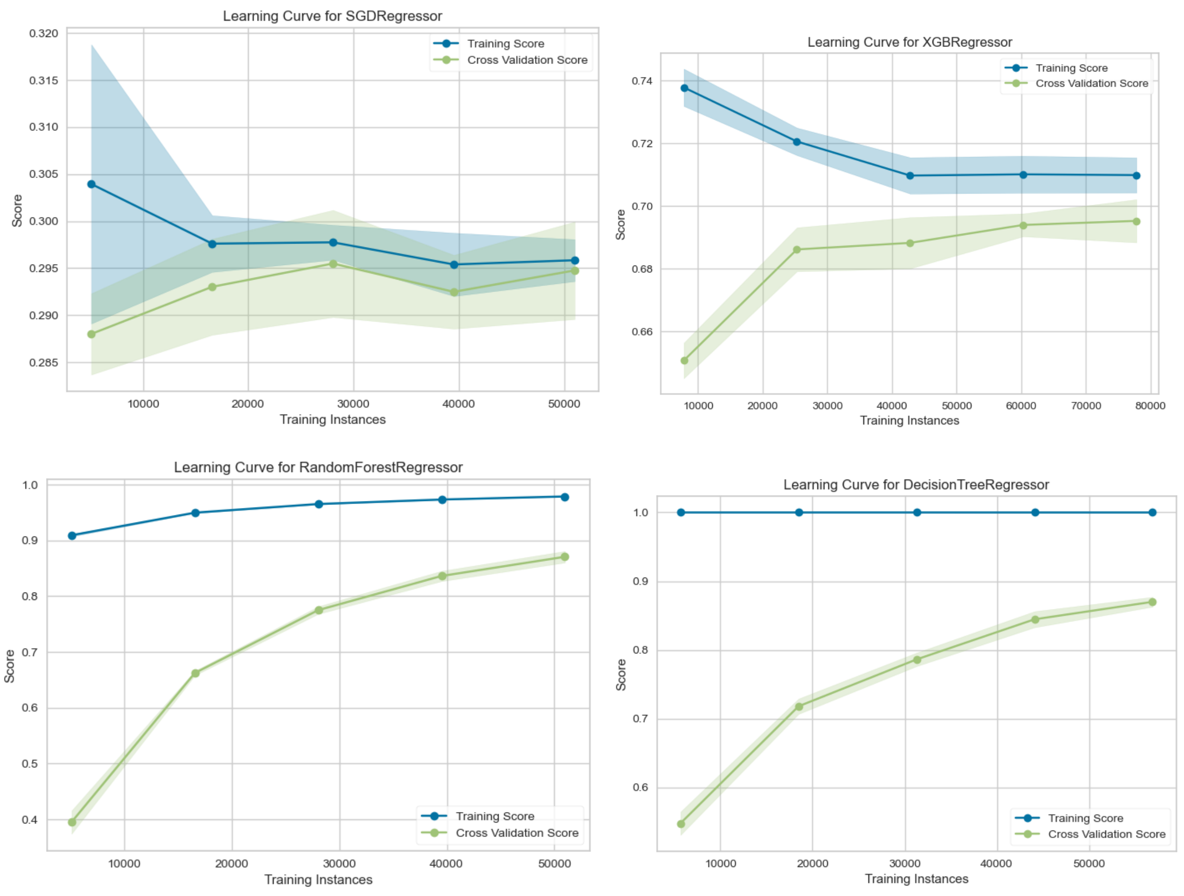


Image 6: learning curves for default [un-regularized] models

The Decision Tree and Random Forest show an overfit training result since they have not been tuned yet. The learning curves were built on default parameter settings.

Regularization

Different learning models use different types of Regularization. The model parameters that I explored are mentioned below.

- Linear Regression with SGD:
    a. penalty: Lasso Regularization (L1) and Ridge Regularization (L2) are considered.

    b. alpha: tunes the regularization term. High value means high regularization.

- Decision Tree, Random Forest and XGBoost:
    a. max_Depth: reduce the maximum depth of the tree to discourage memorization
    b. n_estimators: reduce number of decision trees to improve generalization
    c. max_depth: specify a value to ensure that $\boldsymbol{E_{in}}$ is not 100 %

Machine Learning Models

Models Consideration Rationale

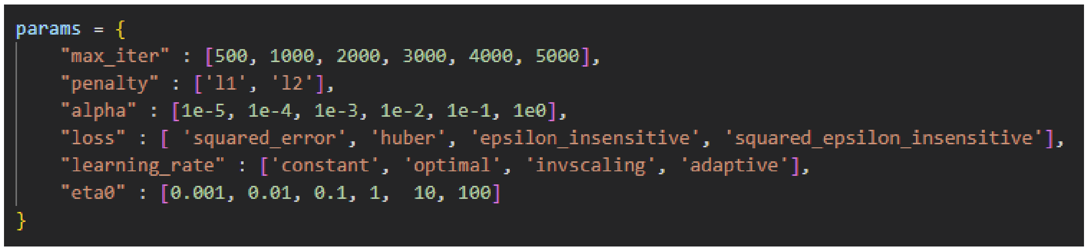
I considered and compared 4 Machine Learning Models for this Learning Problem:

1. Linear Regression with Stochastic Gradient Descent
2. Decision Tree
3. Random Forest
4. XG Boost

The rationale behind this was to implement the more modern and advanced regression tools and compare it to the classic machine learning models we learned through the course. I decided to not consider Support Vector Regressor for this particular learning problem. We are dealing with datasets of sizes that exceed what SVR can handle (~ 20,000 points). Hence, SVR is expected to perform poorly while being computationally demanding. I was able to use Neural Networks for regression. However, tuning in to get the best model was fairly time consuming. I have added a fer initial results in the Appendix. But I have not considered it as a potential model for this study.

Linear Regression with SGD

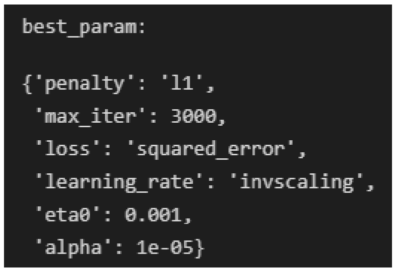
I have used stochastic gradient descent to build a linear regressor. The gradient of the loss is estimated for each sample at a time and the model is updated along the way with a decreasing strength schedule. I use sklearn's SGDRegression function to implement this model. It allowed me to tune several parameters like regularization penalty, loss type, number of iterations and learning rate. The default parameters gave reasonable results. This was a good starting point to continue with hyperparameter tuning. I build a dictionary called params. Below, you can find a snippet of the parameters I considered for tuning.



```
params = {
    "max_iter" : [500, 1000, 2000, 3000, 4000, 5000],
    "penalty" : ['l1', 'l2'],
    "alpha" : [1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1e0],
    "loss" : [ 'squared_error', 'huber', 'epsilon_insensitive', 'squared_epsilon_insensitive'],
    "learning_rate" : ['constant', 'optimal', 'invscaling', 'adaptive'],
    "eta0" : [0.001, 0.01, 0.1, 1,  10, 100]
}
```

Image 7: SGD parameter dictionary

The best score comes for the following parameter configuration:



```
best_param:

{'penalty': 'l1',
 'max_iter': 3000,
 'loss': 'squared_error',
 'learning_rate': 'invscaling',
 'eta0': 0.001,
 'alpha': 1e-05}
```

Image 8: SGD best parameters

The summary of R^2scores and MSE values can be found in the table below.

Table 3: SGD Evaluation Metrics for Translational Velocity (v)

SGD		**Translational Velocity (v)**			
		[Box_Set]		**[Corridor_Set]**	
Evaluation Metric		E_{in}	E_{out}	E_{in}	E_{out}
before tuning	MSE	0.0964	0.0965	0.0915	0.1115
	R^2	0.2736	0.2342	0.2992	0.1932
	MSE	0.1	0.0949	0.0925	0.1121

after tuning	R^2	0.246	0.2468	0.2917	0.1888

Table 4: SGD Evaluation Metrics for Angular Velocity (w)

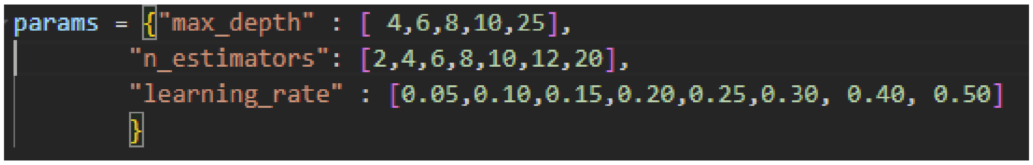
SGD		**Angular Velocity (w)**			
		[Box_Set]		**[Corridor_Set]**	
Evaluation Metric		E_{in}	E_{out}	E_{in}	E_{out}
before tuning	MSE	0.2679	0.3566	0.2277	0.2409
	R^2	-1.0112	0.7738	-0.7511	-0.744
after tuning	MSE	0.0969	0.0972	0.0933	0.1122
	R^2	0.2722	0.2288	0.2828	0.188

After tuning, both the translational and angular velocities gave positive coefficient of determination, improving on what the default model parameters could offer. The general trend suggests that adding regularization, increasing iterations and using a constant learning rate eta0 improves the model. While the E_{in} value falls marginally, the overall $E_{out}$is better. Due to regularization, the estimator learns a simpler model and prevents overfitting. This explains why the model does better on unseen data.

XGBoost

XGBoost is a powerful approach for building supervised regression models. The library is focused on computational speed and model performance. The instance is XGBRegressor and it leverages a regularizing gradient boosting.

XGBoost has a multitude of hyperparameters, but being relatively unexperienced with this learning tool, I only tune three major parameters: maximum depth, number of estimators and learning rate.

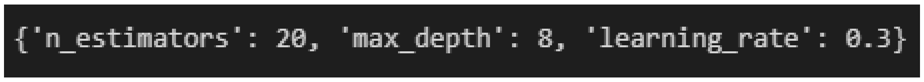


```
params = {"max_depth" : [ 4,6,8,10,25],
          "n_estimators": [2,4,6,8,10,12,20],
          "learning_rate" : [0.05,0.10,0.15,0.20,0.25,0.30, 0.40, 0.50]
          }
```

Image 9: XG Boost parameters dictionary

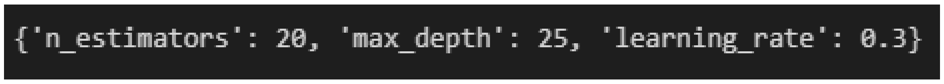
The best parameters were as follows:

Box test set (v):



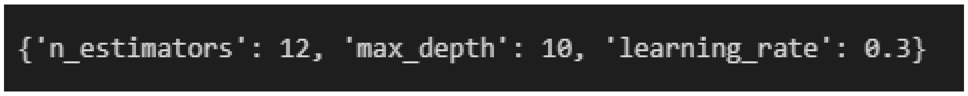
```
{'n_estimators': 20, 'max_depth': 8, 'learning_rate': 0.3}
```

Box test set (w):



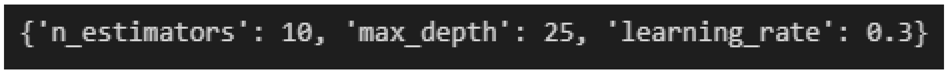
```
{'n_estimators': 20, 'max_depth': 25, 'learning_rate': 0.3}
```

Corridor test set (v):



```
{'n_estimators': 12, 'max_depth': 10, 'learning_rate': 0.3}
```

Corridor test set (w):



```
{'n_estimators': 10, 'max_depth': 25, 'learning_rate': 0.3}
```

You can find the evaluation metric scores on both test sets in Table below. There was a significant improvement in the E_{out} value. The major change from the default parameters was an increase in the number of estimators and a decrease in learning rate. Increasing the number of estimators or tree is good as it encourages gradient boosting.

Table 5: XG Boost Evaluation Metrics for Translational Velocity (v)

XG Boost		**Translational Velocity (v)**			
		[Box_Set]		**[Corridor_Set]**	
Evaluation Metric		E_{in}	E_{out}	E_{in}	E_{out}
before tuning	MSE	0.0289	0.0285	0.0296	0.105
	R^2	0.779	0.7738	0.7728	0.2397
after tuning	MSE	0.01	0.0107	0.0109	0.0906
	R^2	0.9232	0.9151	0.918	0.3441

Table 6: XG Boost Evaluation Metrics for Angular Velocity (w)

XG Boost		**Angular Velocity (w)**			
		[Box_Set]		**[Corridor_Set]**	
Evaluation Metric		$\boldsymbol{E_{in}}$	$\boldsymbol{E_{out}}$	$\boldsymbol{E_{in}}$	$\boldsymbol{E_{out}}$
before tuning	MSE	0.0432	0.0503	0.0361	0.0849
	R^2	0.3835	0.439	0.4932	-0.5082
after tuning	MSE	0.0187	0.0206	0.0157	0.0974
	R^2	0.733	0.7699	0.7797	-0.7305

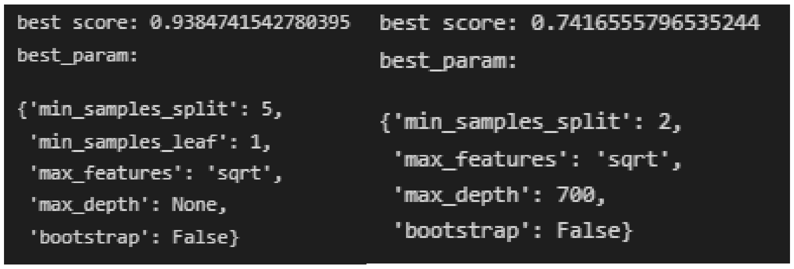
.

Random Forest

Random Forests produce better results, work well on large datasets, and can work with missing data by creating estimates for them. However, they pose a major challenge that is that they can't extrapolate outside unseen data. Knowing this drawback, I wanted to try this model and explore its limitations through practice.

Random forest is an ensemble of decision trees. This is to say that many trees, constructed in a certain "random" way form a Random Forest. Each of the trees makes its own individual prediction. These predictions are then averaged to produce a single result. The averaging makes a Random Forest better than a single Decision Tree hence improves its accuracy and reduces overfitting.

The default results for translational and angular velocity predictions are surprisingly good on Box test set. For Corridor test set however, the E_{out} for v was poor and the E_{out} for w was negative. I separately ran Randomized Search algorithm for v and w. The optimal parameters are shown below:



```
best score: 0.9384741542780395
best_param:

{'min_samples_split': 5,
 'min_samples_leaf': 1,
 'max_features': 'sqrt',
 'max_depth': None,
 'bootstrap': False}
```

```
best score: 0.7416555796535244
best_param:

{'min_samples_split': 2,
 'max_features': 'sqrt',
 'max_depth': 700,
 'bootstrap': False}
```

Image 11: Random Forest best parameters

For Box test set, the new E_{out}did not show any significant improvement on the default values. The $E_{out}$for Corridor test set did increase but not by a large margin.

Table 7: Random Forest Evaluation Metrics for Translational Velocity (v)

Random Forest		**Translational Velocity (v)**			
		[Box_Set]		**[Corridor_Set]**	
Evaluation Metric		$\boldsymbol{E_{in}}$	$\boldsymbol{E_{out}}$	$\boldsymbol{E_{in}}$	$\boldsymbol{E_{out}}$
before tuning	MSE	0.0005	0.0015	0.0005	0.1166
	R^2	0.9963	0.9882	0.9964	0.1562
after tuning	MSE	0.0062	0.0073	0.0065	0.1119
	R^2	0.9524	0.9419	0.9505	0.1902

Table 8: Random Forest Evaluation Metrics for Angular Velocity (w)

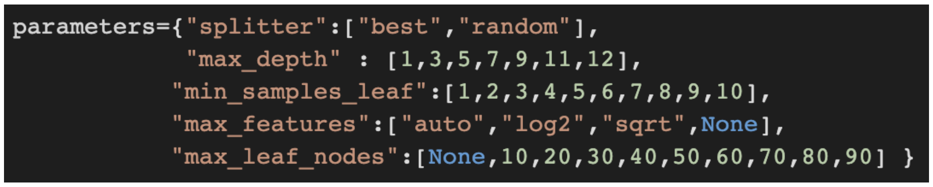
Random Forest		**Angular Velocity (w)**			
		[Box_Set]		**[Corridor_Set]**	
Evaluation Metric		$\boldsymbol{E_{in}}$	$\boldsymbol{E_{out}}$	$\boldsymbol{E_{in}}$	$\boldsymbol{E_{out}}$
before tuning	MSE	0.0012	0.0043	0.0012	0.0969
	R^2	0.9826	0.9522	0.9822	-0.721
after tuning	MSE	0.0969	0.0972	0.0166	0.0727
	R^2	0.7722	0.8032	0.763	-0.2913

It is known that Random Forest Regressor is not great at extrapolating values that lie outside the training set range. When faced with such a scenario, the regressor assumes that the prediction will fall close to the maximum value in the training set. This occurrence is expected to bring the worst out of a regression model like Random Forest. Hence, justifying the poor performance on Corridor test set.

Decision Tree

I try using a Decision Tree to learn linear regressions that can approximately fit a sine curve. The initial results were not too positive. Nevertheless, I go ahead with tuning the model and trying to get the best version of it. Decision tree function allows a multioutput regression; hence, it was relatively quicker to build a model that can learn both v and w.

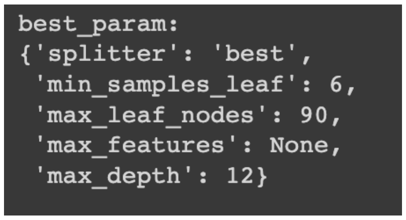
I tune the basic parameters like minimum sample leaf, maximum leaf nodes, and maximum depth. The range explored for each can be seen in snippet below, showing the parameters dictionary.



```
parameters={"splitter":["best","random"],
             "max_depth" : [1,3,5,7,9,11,12],
            "min_samples_leaf":[1,2,3,4,5,6,7,8,9,10],
            "max_features":["auto","log2","sqrt",None],
            "max_leaf_nodes":[None,10,20,30,40,50,60,70,80,90] }
```

Image 12: Decision Tree parameters dictionary

The best parameter values are as follows:



```
best_param:
{'splitter': 'best',
 'min_samples_leaf': 6,
 'max_leaf_nodes': 90,
 'max_features': None,
 'max_depth': 12}
```

Image 13: Decision Tree best parameters

The metric scores can be seen in the table below:

Table 9: Decision Tree Evaluation Metrics

Decision Tree	Translational (v) and Angular (w) Velocity	
	[Box_Set]	[Corridor_Set]

Evaluation Metric		E_{in}	E_{out}	E_{in}	E_{out}
before tuning	MSE	0.0001	0.0015	0.0001	0.2106
	R^2	0.9963	0.9546	0.9992	-1.8196
after tuning	MSE	0.0444	0.0502	0.0573	0.0879
	R^2	0.4883	0.4993	0.3556	0.038

We can see that if the maximum depth of the tree (controlled by the max_depth parameter) is set too high, the decision trees learn too fine details of the training data and tends to overfit. Tuning does not help the learning much and overall, the model performance is still quite poor, both for Box test set and Corridor test set.

Conclusions

It was a good idea to tune every single model picked for the learning problem. In most cases, hyperparameter tuning reduces overfitting and therefore reduces the E_{out} value when running predictions on test sets. Comparing the performance of all the models used, it can be concluded that XG Boost works best for this specific learning problem.

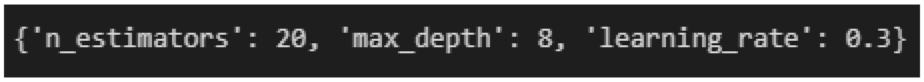
In general, regression trees worked better than linear regression models. Random Forest and XG Boost outperform the Stochastic Gradient Descent based Linear Regressor. XG Boost gives the best R^2 scores while giving the lowest MSE values. The Out of Sample errors are shown below.

Table 10: E_{out} [MSE and R^2] values for XG Boost

XG Boost	**Translational Velocity (v)**		**Angular Velocity (w)**	
	Box test set	**Corridor test set**	**Box test set**	**Corridor test set**
MSE	0.01	0.09	0.02	0.10
$\boldsymbol{R^2}$	0.92	0.34	0.77	-0.73

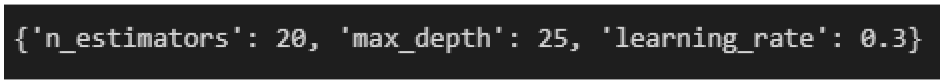
The parameters that worked the best are shown below:

Box test set (v):



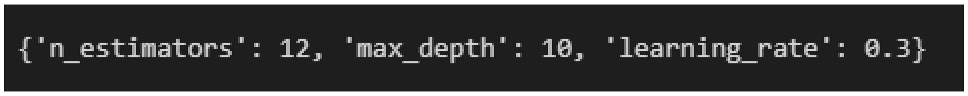
```
{'n_estimators': 20, 'max_depth': 8, 'learning_rate': 0.3}
```

Box test set (w):



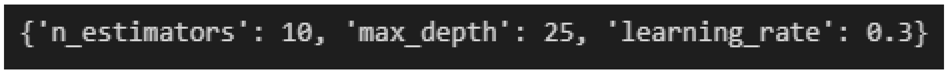
```
{'n_estimators': 20, 'max_depth': 25, 'learning_rate': 0.3}
```

Corridor test set (v):



```
{'n_estimators': 12, 'max_depth': 10, 'learning_rate': 0.3}
```

Corridor test set (w):



```
{'n_estimators': 10, 'max_depth': 25, 'learning_rate': 0.3}
```

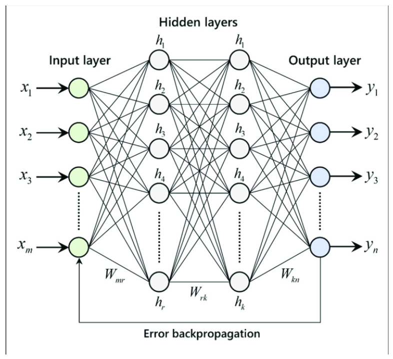
Aside from being the best performing model, XG Boost was also quickest to train on large datasets. And unlike a classic Gradient Boosting Method, XGBoost uses advanced regularization (L2 in this case) to encourage a more generalized machine learning model.

Other models are indeed easier to understand and implement. However, the immense amount of data, and the complexity of the learning problem required a learning architecture that is efficient and robust. Hence, XG Boost came out as the winner. In the future, a more extensive hyperparameter tuning, combined with a larger training data set, could be used to augment the model's capabilities even further.

Appendix

Other Attempted Models: Neural Networks

The Artificial Neural Networks are often used for classification. But they can be used for regression too. I used TensorFlow's Sequential method to build layers. The Neural Network mostly contains several Fully Connected Layers, or Dense Layers. I have used ReLU as the activation function for hidden layers. The final layer uses a linear activation function. The NN model I built contains 3 hidden layers, of 256 neurons each. Instead of the classical stochastic gradient descent, I used the Adam Optimizer algorithm to update network weights.

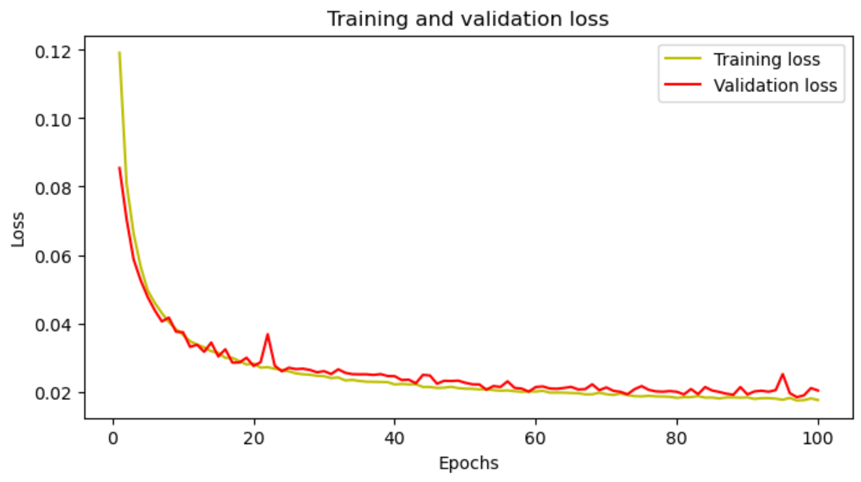


The validation split is still 20%. The hyperparameters to tune here are:

- Number of Epochs
- Number of Hidden Layers
- Learning Rate
- Mini-Batch Size [32, 64, 128, 256]

Regularization used [L1 or L2 or DropOut]

Training Loss Plot – Default Parameters


Training and validation loss
Training loss
Validation loss
Loss
Epochs