CSE 251B Project Milestone Report

Andrew Xi
anxi@ucsd.edu

Kenneth Vuong
k5vuong@ucsd.edu

Zerlina Lai
z6lai@ucsd.edu

1 Task Description and Exploratory Analysis

1.1 Problem A

The goal of this task is to predict the future trajectory of a specific agent (the ego vehicle in our case) in a real world driving scene. Each scene contains multiple agents (pedestrians, different vehicles, etc), and we are provided with the first few seconds of their motion history. Our model uses this information to forecast the future positions of the ego vehicle.

Let N be the number of training scenes, A be the number of agents per scene, T_{obs} be the number of time steps for the training set, and d be the number of features per agent per time step. Then the input tensor for training would be $X \in \mathbb{R}^{N \times A \times T_{obs} \times d}$ and the input tensor for one specific scene would just be $X \in \mathbb{R}^{A \times T_{obs} \times d}$. Each scene, X_i, includes 50 agents with their position (x and y coordinates), velocity (for both x and y direction), heading (heading angle of the agent in radians), and object type. These agents can be from the following object types: vehicle, pedestrian, motorcyclist, cyclist, bus, static, background, construction, riderless bicycle, and unknown.

We aim to predict the ego-vehicle's (agent 0) future trajectory over the next 6 seconds, which corresponds to T_{pred} = 60 time steps. Therefore, the output would be $Y \in \mathbb{R}^{N \times T_{pred} \times 2}$ overall or $Y \in \mathbb{R}^{T_{pred} \times 2}$ for one specific scene. Our prediction task is to learn a function f, such that

$$f : \mathbb{R}^{N \times T_{\text{obs}} \times 2} \to \mathbb{R}^{T_{\text{pred}} \times 2}$$

In our experiment, we have N = 10,000, A = 50, T_{obs} = 50, and T_{pred} = 60. For this task, we use mean squared error as the loss function:

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

This task is important because as autonomous driving becomes more prevalent, it is critical to ensure that they don't create dangerous situations on the road. In real world settings, different agents on the road move in complex and dynamic ways. Autonomous vehicles need to predict others' movements to make safe decisions, such as yielding, turning, lane changing, and stopping to avoid collisions, plan optimal routes, and navigate complex scenarios. By solving this task, we can better understand which models and features contribute to reliable motion prediction for real-world driving.

1.2 Problem B

The training dataset comes from a modified version of the Argoverse 2 dataset. It consists of 10,000 driving scenes, where each scene includes 50 agents observed over 11 seconds. Since the data is sampled at 10 Hz, this results in 110 time steps per agent for each scene. At each time step, each agent has 6 features: x and y positions, x and y velocities, the heading angle in radians, and an object type.

For model training, only the first 50 time steps are used as input, resulting in an input tensor of shape $X \in \mathbb{R}^{10,000\times50\times50\times6}$. The prediction target is the future trajectory of the ego vehicle (agent at index 0) for the next 60 time steps (6 seconds). Each time step in the prediction involves an x and y position, forming an output tensor of shape $Y \in \mathbb{R}^{10,000\times60\times2}$.

The test set has the same input format as the training set, consisting of 2,100 scenes. It has an input of shape $X \in \mathbb{R}^{2,100\times50\times50\times6}$. The expected model output for the test set is a tensor of shape $Y \in \mathbb{R}^{2,100\times60\times2}$. For this specific task, we reshape the output tensor to 126,000 x 2, combining the 60 time steps (6 seconds) of predicted trajectories of the ego agent across all 2,100 scenes into a singular 2D matrix.

Visualizing all position occurrences across all object types reveals the structure of the environment: roads are distinctly visible, with larger roads having higher frequencies of agent observations.

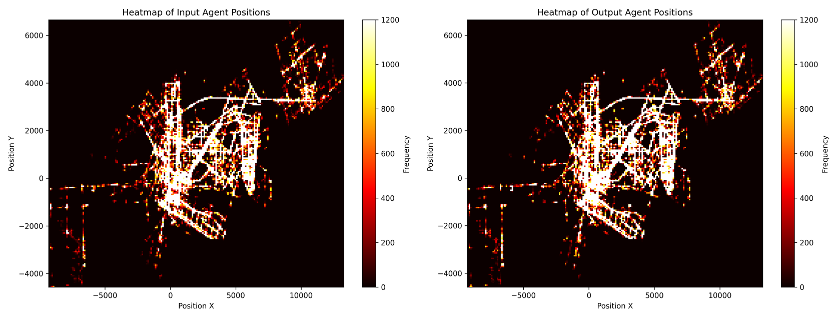


Figure 1: Heatmap of X and Y positions across all agents in the input (first 50) timesteps, and output (last 60) timesteps.

We broke this distribution down by object type as well, to see if there were significant differences in where each object type was commonly observed.

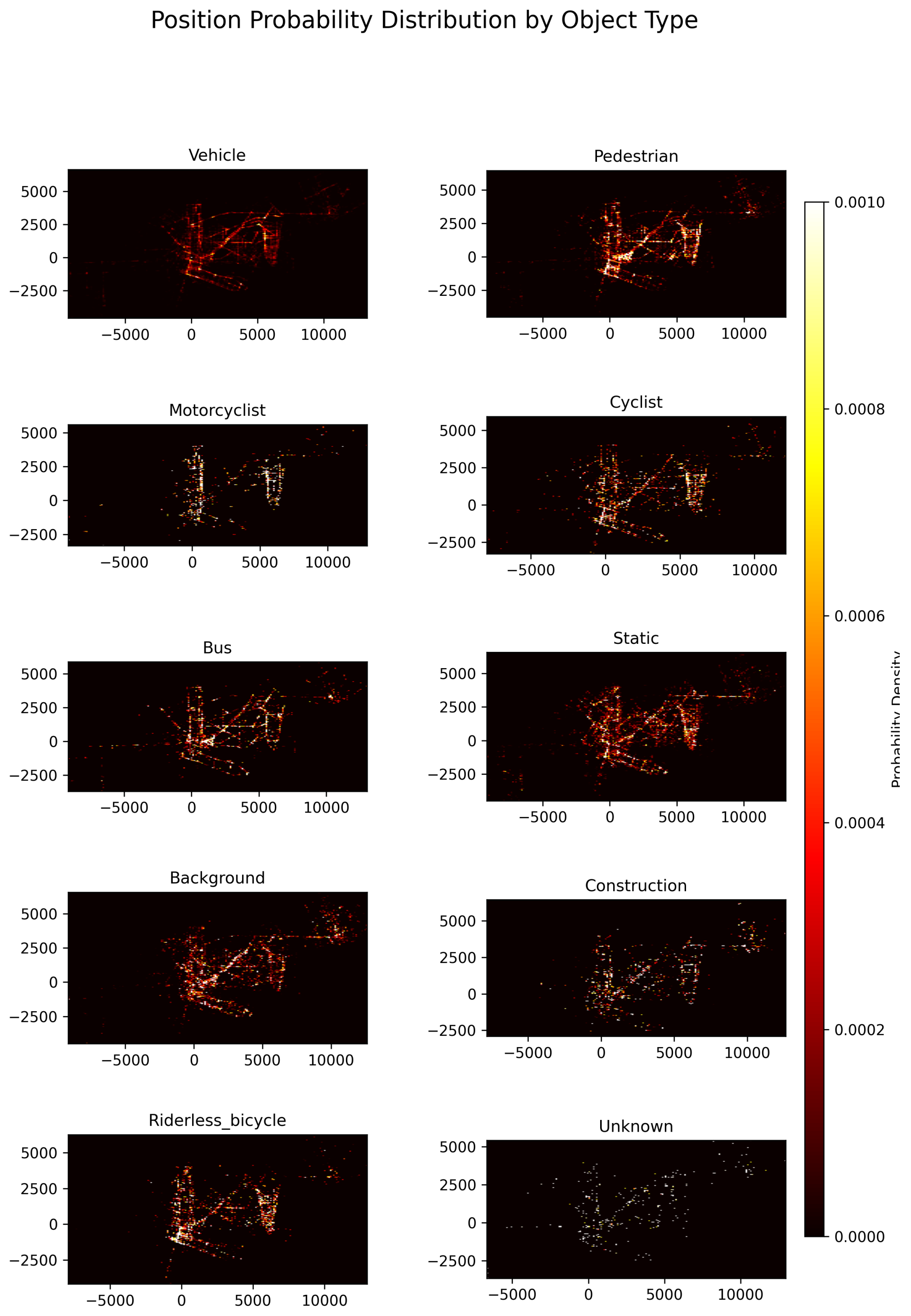


Figure 2: Heatmaps showing the probabilites that an instance of an object class was observed at any particular X,Y location, across all timesteps.

As shown in the above figure, different object classes showed significantly varying distributions, providing some key insights about the environment. For example, the pedestrian, cyclist, and riderless bicycle mappings highlight areas of the environment that are likely more urban, low-speed, and walkable. In contrast, vehicles are distributed fairly evenly over a large number of roads.

We also examined the distribution of headings for each object type.

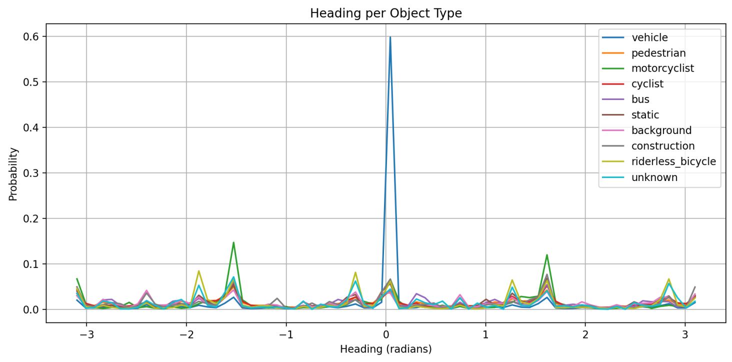
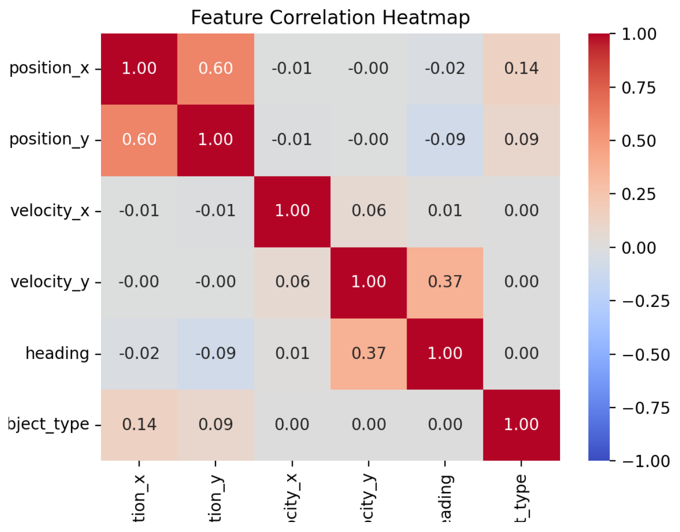


Figure 3: Histogram of headings, in radians, for each object type.

There were a large number of vehicle observations with heading 0, which suggests that these may be the ego vehicles, and that the coordinate systems are defined relative to the ego vehicles. The spikes in observed headings at 1.6 and -1.6 radians is reasonable because these correspond to objects traveling in a perpendicular direction compared to the ego vehicle. This occurs commonly at intersections. The spikes at 0 and +/- 3.14 radians correspond to objects traveling on the same road, in the same and opposite directions respectively.

Lastly, we also did a feature correlation analysis to see if there were any obvious correlations that could help us develop our model. However, we did not find any significantly meaningful pairwise relationships between these features.



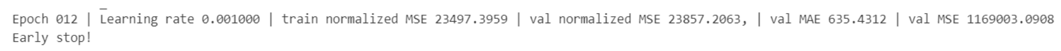
	position_x	position_y	velocity_x	velocity_y	heading	object_type
position_x	1.00	0.60	-0.01	-0.00	-0.02	0.14
position_y	0.60	1.00	-0.01	-0.00	-0.09	0.09
velocity_x	-0.01	-0.01	1.00	0.06	0.01	0.00
velocity_y	-0.00	-0.00	0.06	1.00	0.37	0.00
heading	-0.02	-0.09	0.01	0.37	1.00	0.00
bject_type	0.14	0.09	0.00	0.00	0.00	1.00

Figure 4: Feature correlation heatmap showing a lack of strong correlations between dataset features.

2 Deep learning Model and Experiment Design

2.1 Problem A

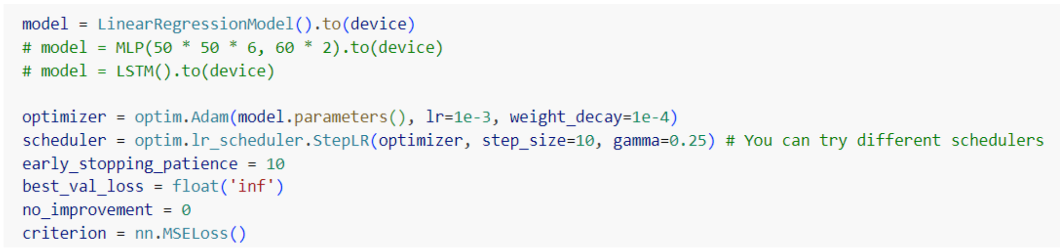
For our deep learning model we utilize the GPU (personal device or Google Colab GPU) to parallelize computation and improve efficiency. For the simple We improve the performance of the provided linear regression model through updating the hyperparameters and optimizers applied to it.
The provided base optimizer is the Adam optimizer, with a learning rate of 0.001 and weight decay of 0.0001. The provided scheduler is the StepLR scheduler with a step size of 20 and a gamma of 0.25. The original performance of the model saw high MSE in both the training and validation set, terminating in twelve epochs with the following errors.



```
Epoch 012 | Learning rate 0.001000 | train normalized MSE 23497.3959 | val normalized MSE 23857.2063, | val MAE 635.4312 | val MSE 1169003.0908
Early stop!
```

Figure 5: Default Linear Regression Model Performance

The initial early stopping suggested that the model performance plateaued faster than what was expected. However, after adjusting the learning rate manually it was found that with smaller learning rates the model does perform better.

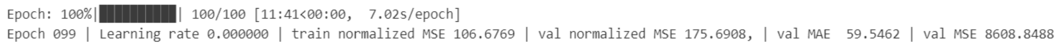


```
model = LinearRegressionModel().to(device)
# model = MLP(50 * 50 * 6, 60 * 2).to(device)
# model = LSTM().to(device)

optimizer = optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-4)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.25) # You can try different schedulers
early_stopping_patience = 10
best_val_loss = float('inf')
no_improvement = 0
criterion = nn.MSELoss()
```

Figure 6: Scheduler Step Size Change

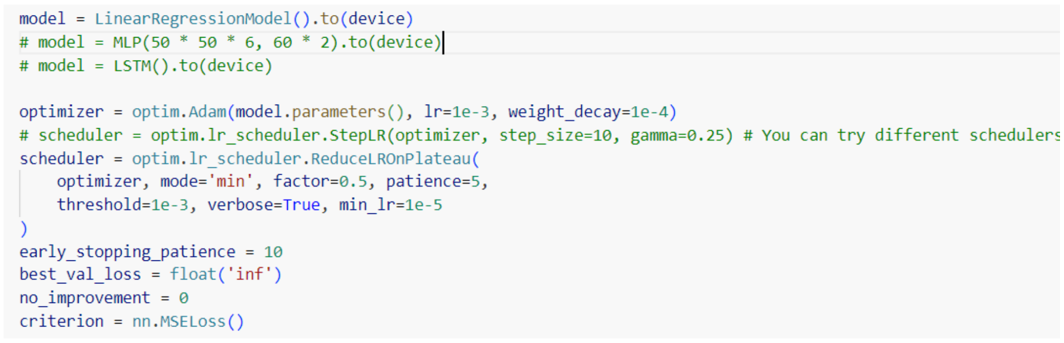
As such, the step size of the scheduler was reduced to 10 to allow for the learning rate to adjust before early stopping. With the error was reduced to a fraction when compared to baseline performance. The model does show consistent performance improvements, as the early stopping trigger was never activated. The final performance is shown below for this strategy is shown below.



```
Epoch: 100%|██████████| 100/100 [11:41<00:00,  7.02s/epoch]
Epoch 099 | Learning rate 0.000000 | train normalized MSE 106.6769 | val normalized MSE 175.6908, | val MAE  59.5462 | val MSE 8608.8488
```

Figure 7: Scheduler Step Size Change Performance

In recognition that the learning rate became too small to make meaningful updates to the model to improve performance, the ReduceLROnPlateau scheduler was implemented. This scheduler dynamically decays the learning rate by observing when validation performance plateaus and scales the learning rate by a factor of 0.5.

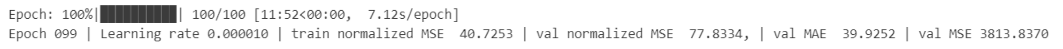


```
model = LinearRegressionModel().to(device)
# model = MLP(50 * 50 * 6, 60 * 2).to(device)
# model = LSTM().to(device)

optimizer = optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-4)
# scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.25) # You can try different schedulers
scheduler = optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode='min', factor=0.5, patience=5,
    threshold=1e-3, verbose=True, min_lr=1e-5
)
early_stopping_patience = 10
best_val_loss = float('inf')
no_improvement = 0
criterion = nn.MSELoss()
```

Figure 8: ReduceLRonPlateau Scheduler

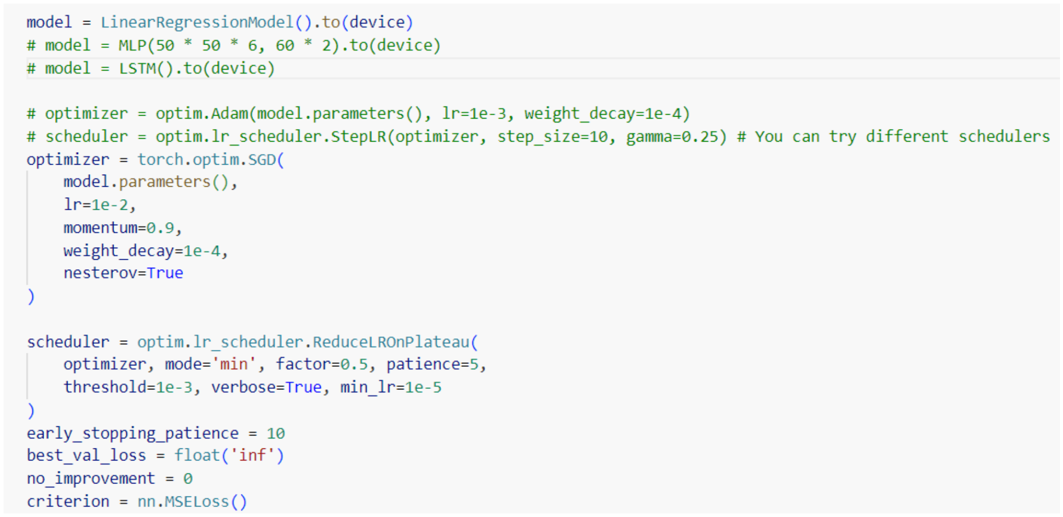
This change improved performance even further, reducing the errors to the following values.



```
Epoch: 100%|██████████| 100/100 [11:52<00:00,  7.12s/epoch]
Epoch 099 | Learning rate 0.000010 | train normalized MSE  40.7253 | val normalized MSE  77.8334, | val MAE  39.9252 | val MSE 3813.8370
```

Figure 9: ReduceLRonPlateau Scheduler Performance

Finally, we experiment with the optimizer. We implement Stochastic Gradient Descent (SGD) with momentum to help converge faster, escape local minima, and improve generalization. We implement a momentum factor of 0.9 as a general factor to smooth gradients meaningfully. Nesterov Momentum is implemented to have greater predictive ability and more meaningful updates.



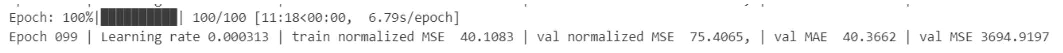
```
model = LinearRegressionModel().to(device)
# model = MLP(50 * 50 * 6, 60 * 2).to(device)
# model = LSTM().to(device)

# optimizer = optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-4)
# scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.25) # You can try different schedulers
optimizer = torch.optim.SGD(
    model.parameters(),
    lr=1e-2,
    momentum=0.9,
    weight_decay=1e-4,
    nesterov=True
)

scheduler = optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode='min', factor=0.5, patience=5,
    threshold=1e-3, verbose=True, min_lr=1e-5
)
early_stopping_patience = 10
best_val_loss = float('inf')
no_improvement = 0
criterion = nn.MSELoss()
```

Figure 10: SGD with Nesterov Momentum

Our final performance is shown below, for SGD with Nesterov Momentum. This suggests that the manually-tuned momentum based updates in SGD were not tuned well enough to beat adaptive learning rates of the Adam optimizer, likely due to better handling of gradient variability and faster convergence. The final performance when compared to the baseline is as follows: 0.16% train normalized MSE, 0.31% validation normalized MSE, 6.30% validation MAE, and 0.31% validation MSE.



```
Epoch: 100%|██████████| 100/100 [11:18<00:00,  6.79s/epoch]
Epoch 099 | Learning rate 0.000313 | train normalized MSE  40.1083 | val normalized MSE  75.4065, | val MAE  40.3662 | val MSE 3694.9197
```

Figure 11: SGD with Nesterov Momentum Performance

For our experiments, we trained the model for up to 100 epochs with early stopping based on the validation loss. For early stopping, training was terminated early if the validation loss did not improve for 10 consecutive epochs. We used a batch size of 32, which provided a good trade-off between training stability and computational efficiency on our GPU platform. For each experiment, the time to train each epoch was typically around 7 seconds.
In our approach towards the following problem, we experimented with Linear, Lasso (L1 regularization), and Ridge Regression (L2 regularization) models through the scikit-learn library. As this library is only compatible with CPUs for these models, we run all training and inference on the system CPU.

2.2 Problem B

The first model we implemented for the trajectory prediction task was a Linear Regression model. Our goal was to start with a simple, interpretable model that allowed us to focus on feature engineering and understand how different motion characteristics contribute to future trajectory prediction. We designed a feature extraction pipeline using only the first 50 time steps of the ego vehicle, as that will be the only information we will be provided with during inference. These features include:

- raw positions, velocities, and heading angles at regular intervals (every 10 time steps) and at the last 5 time steps
- average velocities and headings overall and across regular intervals
- estimated accelerations at regular intervals
- total distance traveled by the ego vehicle
- distances to the five nearest surrounding agents at regular intervals
- object type

It is important to note that not all of these features were used, depending on how well they contributed to the loss.

While training loss was low, the test loss revealed signs of overfitting. To address this, we experimented with Lasso (L1 regularization) and Ridge Regression (L2 Regularization) to apply regularization and improve robustness. These methods allowed the model to either penalize large coefficients or eliminate irrelevant features altogether. To optimize performance, we used cross-validation to tune the regularization strength.

3 Experiment Results and Future Work

3.1 Training loss: MAE

Since we have so far only fitted regression models, we don't have training steps in the same sense that neural networks do. However, we can still visualize the MAE over each k-fold cross validation step. We expect the MAE to be similar across all folds if the data is independent and identically distributed, which we do observe.

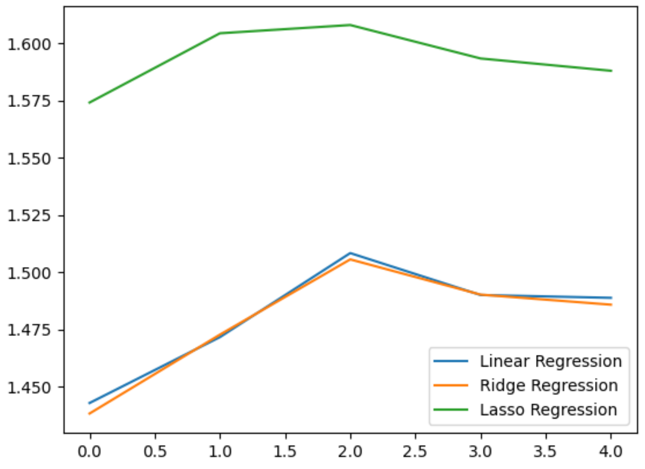


Figure 12: Plot showing MAE for each split in 5-fold cross validation training for linear, ridge, and lasso regression models.

3.1.1 Model Performance Comparison

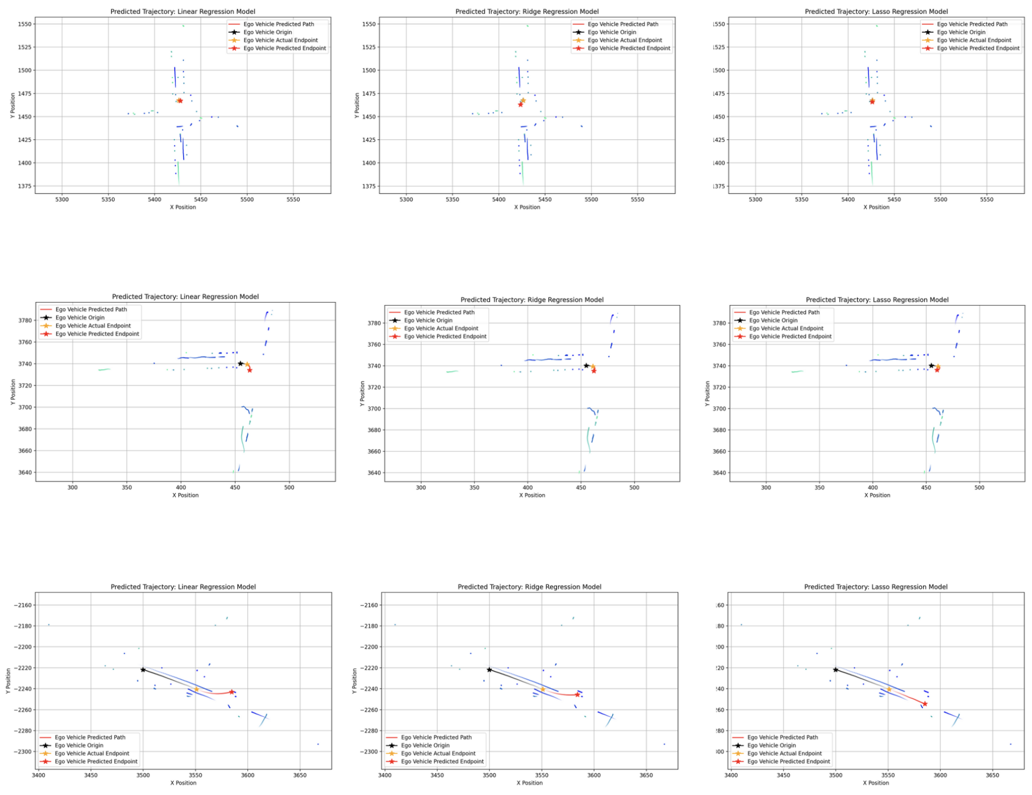


Figure 13: Plots of predicted vs actual trajectories for a set of randomly selected frames, compared between the Linear, Ridge, and Lasso regression models.

We visually examine a randomly chosen set of predictions from the test set, from each of the three models (Linear Regression, Ridge Regression, and Lasso Regression). Overall, the Lasso model appears to have the most reasonable predicted trajectories, and the predicted position of the ego vehicle at the final timestep is closest to the actual position for this model. In the final row of frames in figure 13, both ridge and linear regression models predict the ego vehicle will swerve into what appears to be a pedestrian or other slow-moving vehicle. Although in actuality, the vehicle stops and does not move, the Lasso model produces the most realistic trajectory out of the three (where the vehicle stays on the road and continues traveling forward).

3.2 Current Ranking

Currently, our Lasso regression model ranks #48 on the leaderboard, with a score of 11.82991.

3.3 Analysis and Improvements

While regression models are easy to interpret and fast to train, they are fundamentally limited in their ability to model complex spatiotemporal dynamics like those present in our dataset. Even though Lasso produced cleaner and more plausible trajectories, its performance still falls short of what can be achieved with more complex models.

Because regression models tend to favor linear predictions, they often fail to capture other behaviors like stopping, turning, or reactions to other agents' actions. This can be observed from the swerving or continued motion predictions from the regression models, especially the linear and ridge models. However, even the Lasso model fails to accurately model scenarios where vehicles stop in reaction to other agents.

In the coming weeks, we plan to experiment with more complex models that are better able to capture spatiotemporal dynamics. RNNs and Transformers, for example, are good candidates for sequence-based data such as this. Another important aspect is interaction modeling, which attention mechanisms and graph neural networks can learn. By moving from simple regression baselines to these more advanced models, we aim to improve both the performance and realism of our predictions.