HW4 –Predictive Modeling

May 21, 2025

1 Introduction

This homework explores a range of predictive models, including traditional machine learning algorithms and neural networks, to estimate housing prices. The performance of these models is assessed through comprehensive statistical analyses and visualization techniques.

2 Data Cleaning

The dataset contains numerical features: beds, full_baths, half_baths, sqft, year_built, sold_price, lot_sqft, county, stories, parking_garage, latitude, longitude; and categorical features: style and county. Firstly, for the numerical features, we did the sanity check. From the sanity check, we found that the feature 'lot_sqft' contains some zero values, which is nonsense, so we transformed them into NaN values. Then we did the missingness analysis, and the results are shown in table 1: We can see the data is good and contains no missing values except lot_sqft, as we transformed 0 into NaN value. Besides, I did the descriptive analysis for all numerical features in the dataset and were shown in the table 2. From the table 2, you can see features such as sqft and parking_garage contain some outliers. This might be related to wrong imputation. In sanity check process, I drew boxplots and did the following steps:

- **sqft:** I dropped the sqft greater than 1.0×10^7
- **parking_garage:** Drop the parking_garage greater than 10.

Besides, I plotted a histogram of the sold_price feature and applied a log transformation due to its approximately exponential distribution. You can see from figure 3.

3 Predictive Analysis

3.1 Linear Regression Model for Housing Price Prediction

We constructed a **Linear Regression** model to predict housing sale prices using structured property features. The modeling process consists of the following steps:

- **Feature preprocessing:**
  - **Numerical features** (e.g., number of beds, baths, square footage) were passed through without transformation.
  - **Categorical features** (e.g., building type, garage type) were encoded using `OneHotEncoder` with `handle_unknown='ignore'` to manage unseen categories during inference.
- **Pipeline construction:** We used *ColumnTransformer* to preprocess features and integrated it with *LinearRegression()* in a unified `Pipeline`.
- **Data split:** The data was partitioned into a training set (80%) and a testing set (20%) using `train_test_split`, and set a fixed validation set with random_state = 0.

The model was evaluated on the testing set, resulting in:

Table 1: Missing Value Counts and Percentages

Column	Missing Count	Missing Percentage
property_url	0	0.00
style	0	0.00
beds	0	0.00
full_baths	0	0.00
half_baths	0	0.00
sqft	0	0.00
year_built	0	0.00
sold_price	0	0.00
lot_sqft	1410	17.33
latitude	0	0.00
longitude	0	0.00
county	0	0.00
stories	0	0.00
parking_garage	0	0.00

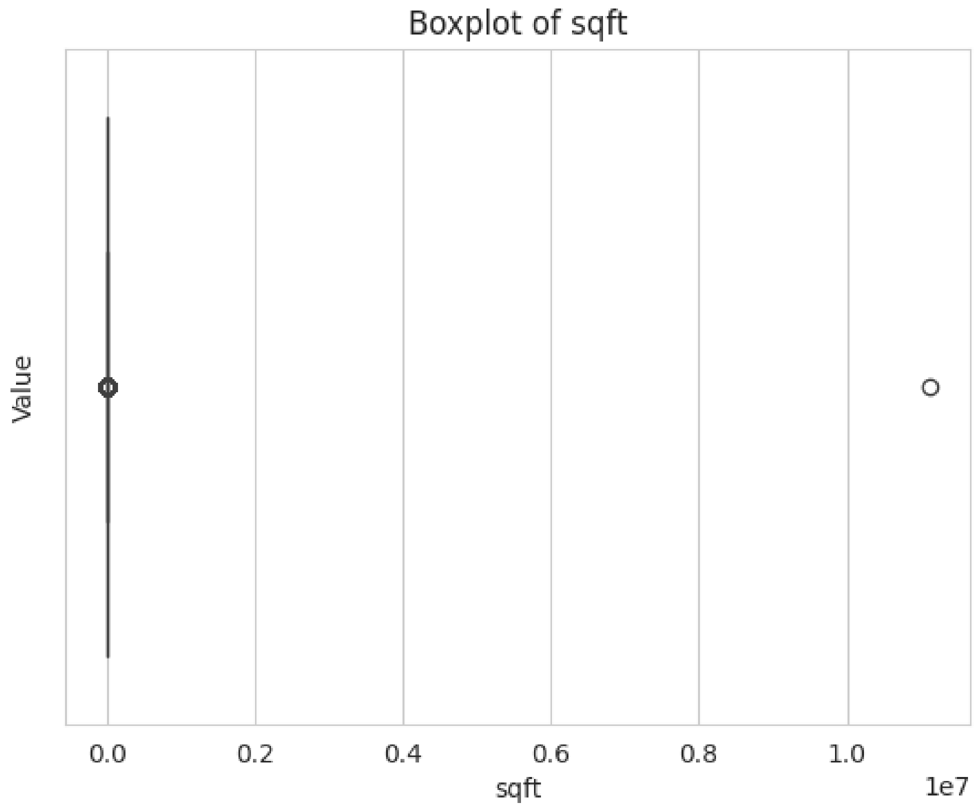

Figure 1: Boxplot for sqft

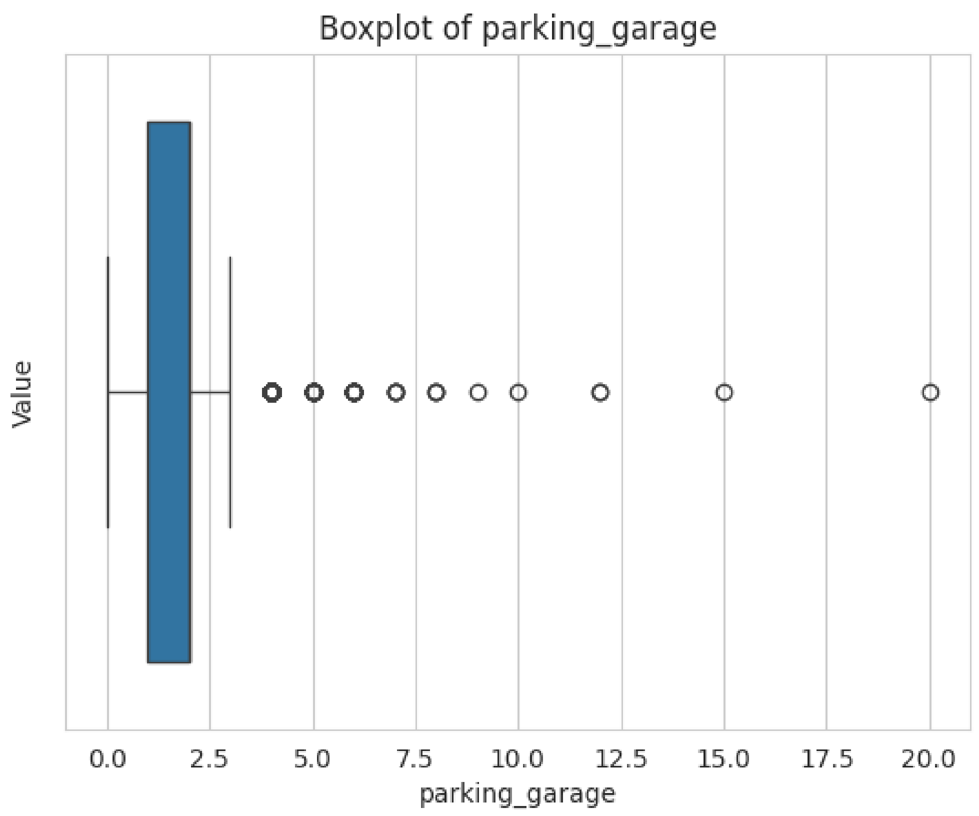

Figure 2: Boxplot for parking_garage

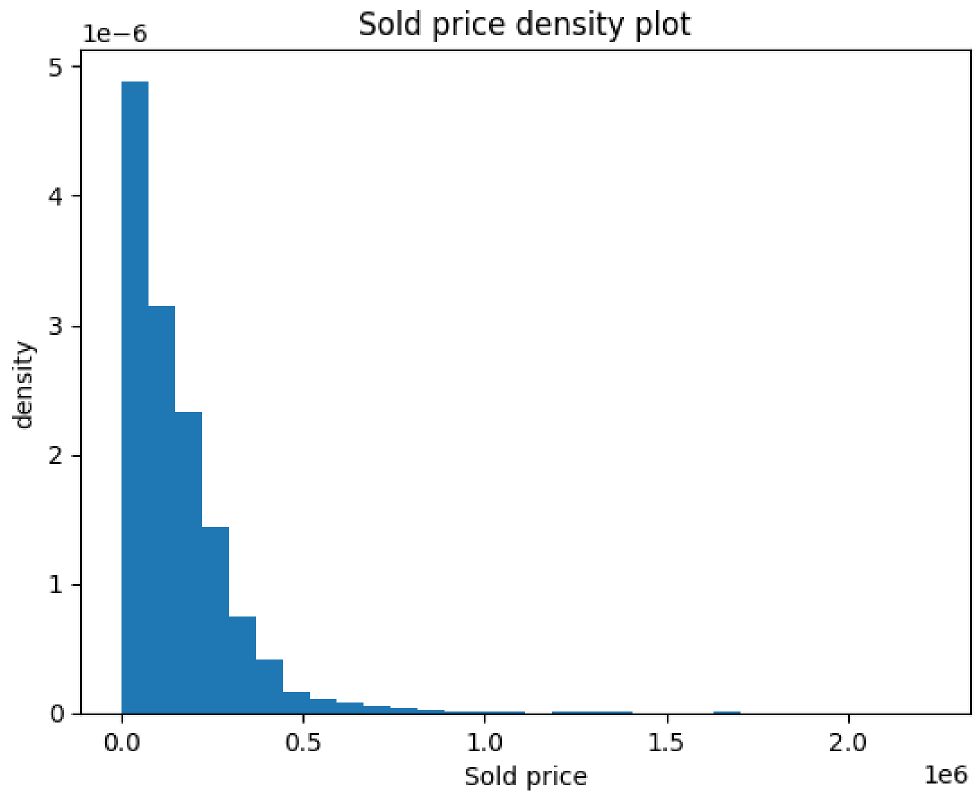

Figure 3: Sold price density plot

Table 2: Descriptive Statistics of Housing Dataset (Selected Features)

	beds	full_baths	half_baths	sqft	sold_price	stories	parking_garage
count	8138.00	8138.00	8138.00	8.14e+03	8.14e+03	8138.00	8138.00
mean	3.07	1.69	0.38	3.04e+03	1.55e+05	1.73	1.52
std	0.97	0.83	0.54	1.23e+05	1.51e+05	2.97	1.06
min	0.00	0.00	0.00	5.00e+01	5.00e+02	0.00	0.00
25%	2.00	1.00	0.00	1.09e+03	4.85e+04	1.00	1.00
50%	3.00	2.00	0.00	1.40e+03	1.18e+05	2.00	2.00
75%	4.00	2.00	1.00	1.94e+03	2.14e+05	2.00	2.00
max	8.00	8.00	9.00	1.11e+07	2.23e+06	95.00	20.00

- **Mean Squared Error (MSE):** 0.4286
- **R-squared ($\mathbf{R^2}$):** 0.6062.

We further visualized the distribution of predicted values using a histogram with 30 bins to understand the spread and bias of the predictions. The results are shown in figure 4 The results show that the predicted values are more concentrated around the center compared to the true values and exhibit a leftward skew relative to the true distribution.

Error terms Analysis: We also plot the histogram of predicted errors shown in figure 5, you can see error terms are approximately normal distribution with mean 0, which corresponds the prerequisite for using the linear regression model.

To improve generalization and address potential multicollinearity among features, we implemented two regularized linear models: **Lasso** (L1 regularization) and **Ridge** (L2 regularization), using five-fold cross-validation for hyperparameter tuning.

- **Lasso Regression (L1):** We applied `LassoCV` with 5-fold cross-validation to automatically select the optimal regularization strength. The modeling pipeline integrates data preprocessing with the Lasso regressor. The model was evaluated on the test set. The Mean Squared Error (MSE) is **0.9580** and R-squared (R^2) is **0.4388**.
- **Ridge Regression (L2):** For L2 regularization, we used `RidgeCV` with a predefined range of regularization strengths $\alpha \in \{0.001, 0.01, 0.1, 1.0, 10.0\}$, again using 5-fold cross-validation. The Mean Squared Error (MSE) is **0.7445** and R-squared (R^2) is **0.5639**.

The performance of both models on the test set shows:

- **Mean Squared Error (MSE):** Linear regression model has lower MSE than ridge and lass regression models, which means linear model performed better than other two models
- **R-squared (R^2):** Linear regression model has higher R^2 than ridge and lass regression models, which means linear model performed better than other two models

Possible Reasons: In datasets where multicollinearity is low, sample size is large, and features are truly informative, these penalties can hurt performance by introducing bias.

3.2 Decision Tree Regression and Model Selection

3.2.1 Model Training

To explore non-linear relationships in the data, we implemented a **Decision Tree Regressor**. We evaluated model complexity by tuning the `max_depth` hyperparameter, which controls the maximum depth of the tree and acts as a regularization mechanism.

- We constructed a pipeline combining the preprocessing step with `DecisionTreeRegressor`, and trained separate models with `max_depth` ranging from 1 to 9.
- For each depth, we recorded the **Mean Squared Error (MSE)** on the test set to evaluate model performance.

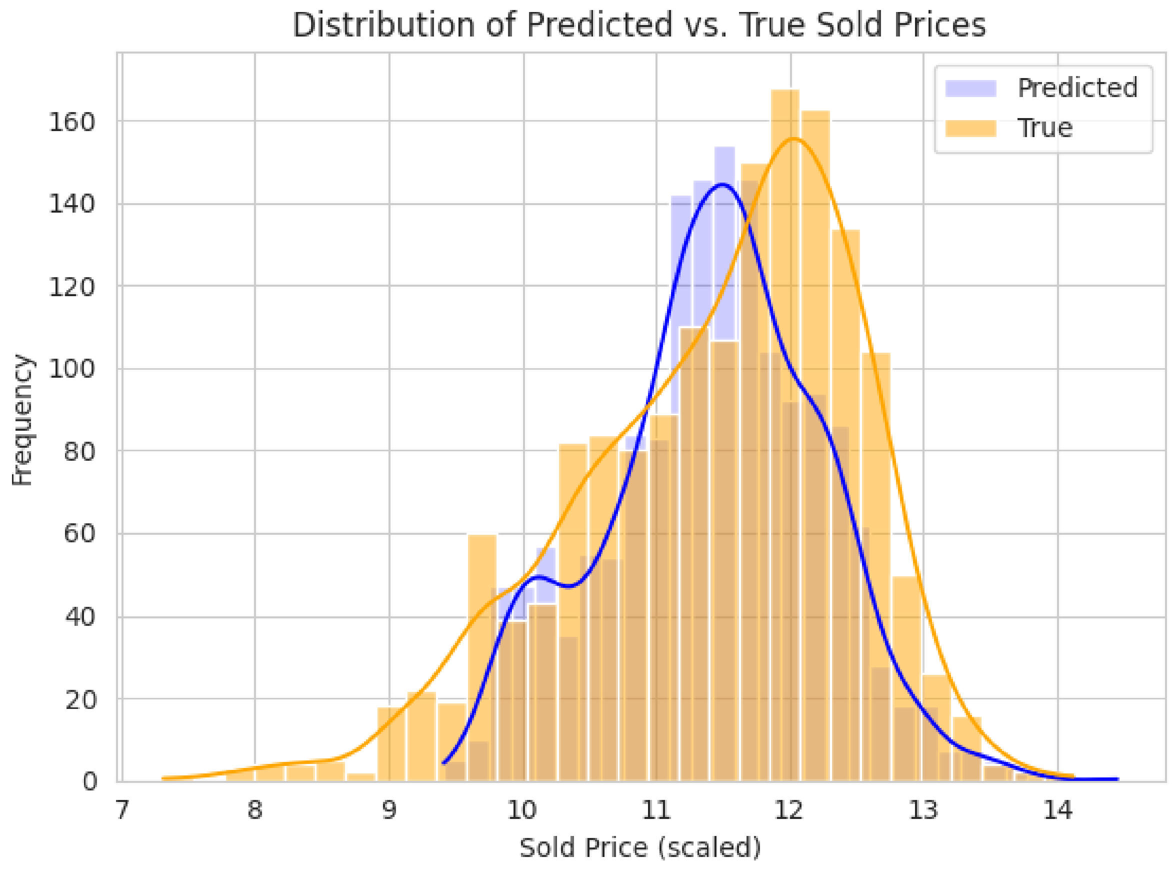

Figure 4: Predictive vs True Linear model

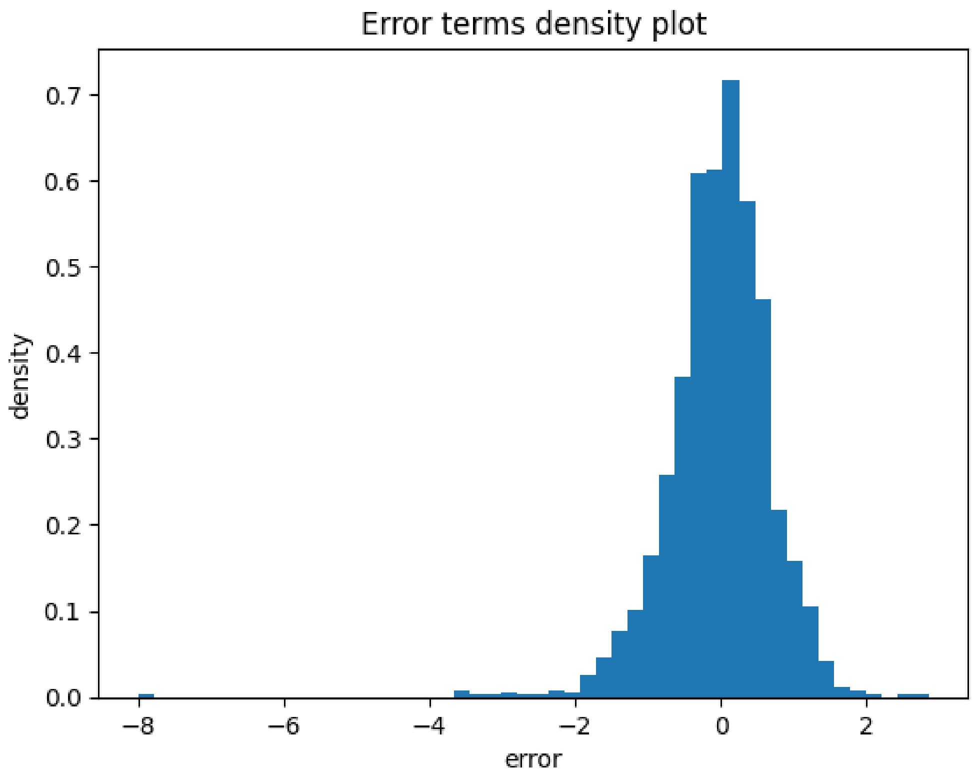

Figure 5: Error terms Density plot

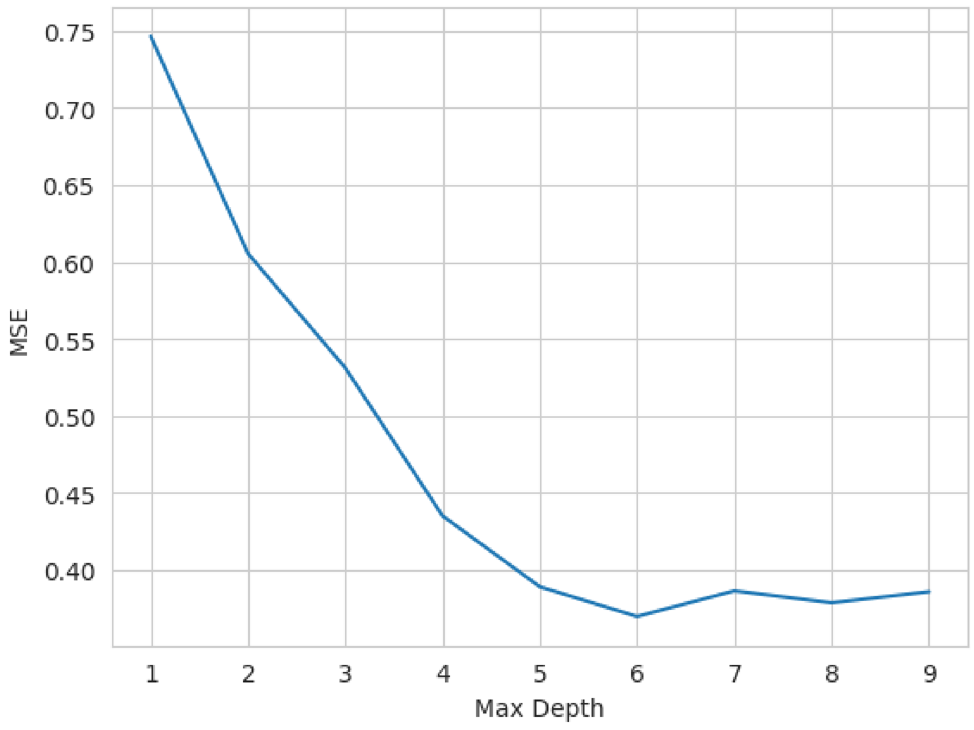

Figure 6: Decision Tree max_depth choice

- The results were plotted to visualize the relationship between tree depth and prediction error(figure 6). The lowest test error was achieved at depth 6, indicating a suitable balance between underfitting and overfitting.

After identifying the optimal tree depth, we trained a final decision tree model using `max_depth = 6` and the MSE for this model is **0.3859**.

3.2.2 Visualization

The fitted model was extracted from the pipeline and the feature names were retrieved using `get_feature_names_out` from the `ColumnTransformer`. This approach allows for interpretable, non-linear regression modeling, and provides a visualizable structure suitable for understanding how input features contribute to predicted outcomes. From the plot, we can predict the sold based on the requirement shown in the plot. Take one specific node as examples,

Decision Tree Interpretation. Figure 19 illustrates a segment of the regression decision tree used to predict housing prices. This tree is built by recursively splitting the data based on feature thresholds that reduce prediction error. Each node represents a decision rule, and the terminal (leaf) nodes show the average predicted value for the samples falling into that category. For a specific node you can see, the root node first splits on `num_parking_garage`:

- **Left branch (num_parking_garage $\leq$ 1.5):** Homes with fewer than or equal to 1.5 garage spaces are separated into this group. These typically represent properties with limited parking. This node includes 692 samples with an average predicted value of 10.10.
  - The next split is based on `num_longitude` $\leq$ -90.07, indicating that geographic location (longitude) plays a significant role in refining the price prediction for homes with limited parking.
- **Right branch (num_parking_garage $>$ 1.5):** Homes with more than 1.5 garage spaces are likely larger or located in more suburban settings. This node contains 393 samples and has a higher average predicted value of 10.52.
  - It then splits on `num_sqft` $\leq$ 1313.5, reflecting the importance of house size in price prediction. Larger square footage is typically associated with higher prices.

Overall, the model uses interpretable feature thresholds—garage capacity, square footage, and geographic location—to partition the data into homogenous subgroups with distinct price predictions. The predicted values at each node represent the average of the target variable (e.g., log-transformed sale price) among the samples that satisfy the decision rules up to that point. This approach enables not only accurate prediction but also transparent reasoning about how different factors influence housing prices. **As the plot is too big, so I controlled the max_depth = 3 and drew the plot (figure 19).**

3.3 Random Forest

3.3.1 Model Training and Tuning

Then I changed the decision tree to a random forest and did the training and evaluation process. I used the n_estimator = 100. The MSE of the result is **0.7631**. And I tuned the two key parameters in the random forest: n_estimators and max_depth. And all results are shown in table 3.

Tune and Improvement: When controlling for the number of estimators, we observe that the MSE loss increases first and then decreases as the maximum depth increases. When controlling for the maximum depth, the MSE loss also decreases with a higher number of estimators. Overall, the maximum depth appears to have a greater impact on the MSE loss compared to the number of estimators because it controls the depth of the trees, which can lead to underfitting or overfitting if not properly tuned. So I think the max_depth is more important than other parameters. And sometimes if we don't control the max_depth, the accuracy is better than when controlling the max_depth for some specific n_estimators. At last, we chose the lowest estimator with **n_estimator = 100 and max_depth = 20**.

Table 3: Random Forest MSE Loss under Different Hyperparameters

n_estimators	**max_depth**	**MSE Loss**
50	None	0.2675
50	10	0.2802
50	20	0.2675
50	30	0.2684
100	None	0.2641
100	10	0.2776
100	20	0.2637
100	30	0.2648
200	None	0.2637
200	10	0.2785
200	20	0.2639
200	30	0.2638

3.3.2 Random Forest Visualization

The plot 7 shows that the result aligns closely with the true y. It appears that the Random Forest predictions are more concentrated and tend to underestimate the actual values. I think the reason behind it is that averaging inherently smooths predictions, reducing variance and causing predictions to cluster near the center of the distribution.

3.4 Neural Networks

We implemented a feedforward neural network using PyTorch for a regression task. The architecture includes one hidden layer with 64 neurons and a sigmoid activation. The model was trained with MSE loss and optimized using Adam with a learning rate of 0.001, over 100 epochs. Inputs and targets were converted to `float32` tensors, and training loss was plotted across epochs and shown in the figure 8. The MSE loss for the test set is 1.0885.

Table 4: Neural Network Model Configuration

Parameter	**Value**
Input Layer Size	2168
Hidden Layer Size	64
Activation Function	Sigmoid
Loss Function	MSELoss
Optimizer	Adam
Learning Rate	0.001
Batch Size	64
Epochs	100
Input Format	`float32` tensors
Output Size	1

We observed that the MSE loss of the neural network was higher than that of the Random Forest and Linear Regression models. This may be due to the neural network's sensitivity to feature scales. Therefore, I hypothesized that standardizing the input features could help reduce the MSE loss and retrained the model accordingly. I used the MinMaxScalar() to do the transformation for all features and trained the model with the same parameters again. The MSE of the test set is **0.0069**. And you can see the training loss in figure 9.

3.4.1 Visualization

I visualized the output distribution of the neural network and found it to be significantly more concentrated than the true target distribution, with predictions clustering around the mean. This suggests the model might be underfitting or overly regularized. After retraining the model using different random seeds, I found that the issue persisted across multiple runs, indicating this is a consistent problem. You can see from figure 10.
Then I added a fixed random seed into the model and trained again. The results are shown in figure 11. The results

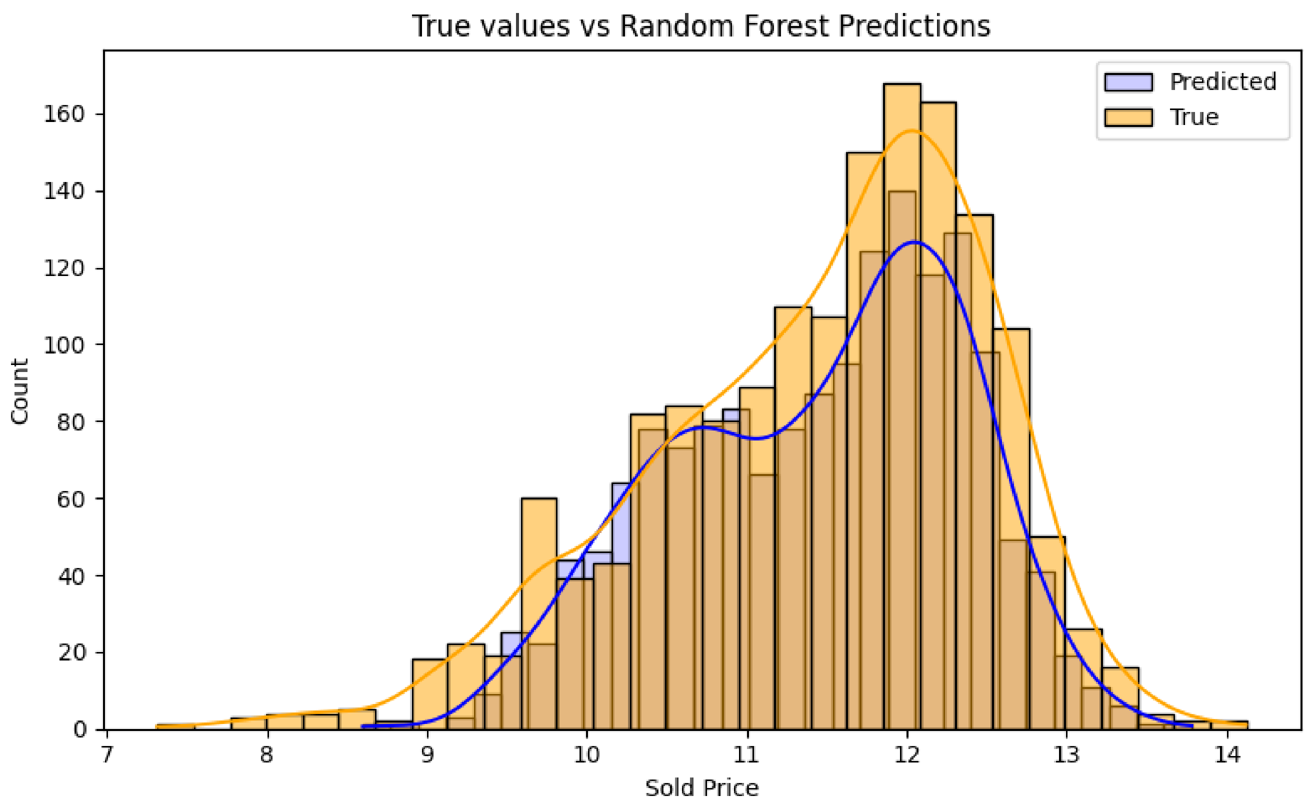

Figure 7: Random Forest Predict vs True y

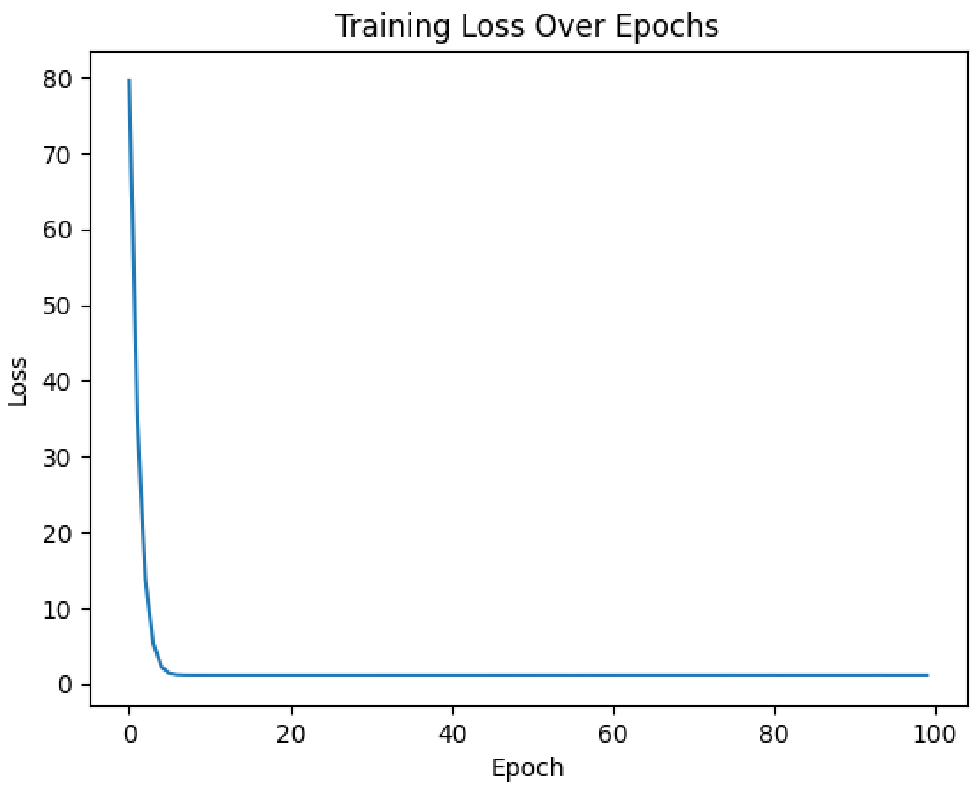

Figure 8: Loss for NN with Sigmoid function

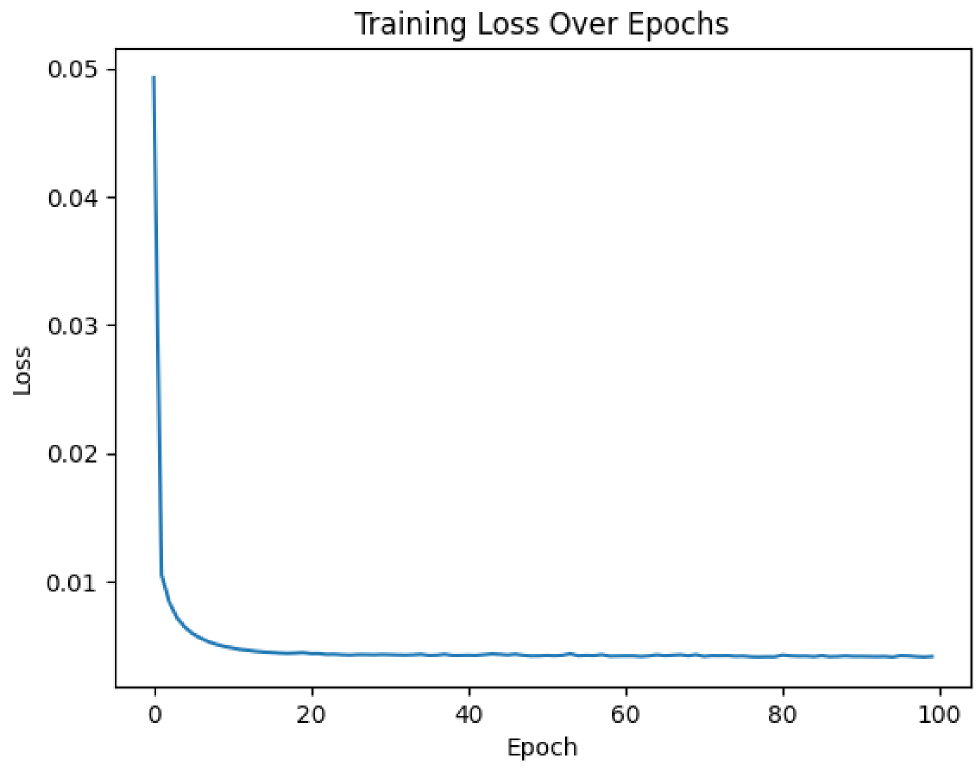

Figure 9: Loss for NN after standardization

solved the concentrated problems, which means **this problem is related to whether we fixed the seeds or not**. I believe the reason is that machine learning involves many sources of randomness. If we don't control the random seed, the model may produce different results each time it's trained — even when using the same data. Besides, I hypothesize that the combination of MSE loss and insufficient model capacity leads to this behavior. I plan to experiment with alternative more robust loss functions (e.g., Huber loss) to address this problem, and the plots are shown in the figure 12. From the plot, we can also infer that **the loss function may be a potential factor contributing to the concentration issue**.

3.4.2 Parameters Tuning

I changed the three ways and drew three loss plots below. I didn't change the learning rate as I found all of them appear converge phenomena when epochs greater than 50. So I think the defaulth learning rate=0.001 is good enough. The benchmark Loss is **0.0069**.

- **Method 1:** Optimizer changes into AdamW, added the L2 norm (weight_decay = 1e-4). Change the Activation into ReLU. The MSE loss is **0.0077**. Show in the figure 13.
- **Method 2:** Layer changed into 5, adding dropout layers(dropout = 0.3). The MSE loss is **0.006723**. Show in the figure 14.
- **Method 3:** Layer changed into 5, but not adding dropout layers. The MSE loss is **0.006754**. Show in the figure 15.
- **Method 4:** Only change the Activation function into ReLU. The MSE loss is **0.0076**. Show in the figure 16.

Compared these models, Method 2 improves the prediction accuracy. Some findings are below:

- Adding layers can help us increase the prediction accuracy, as method 3 is better than the baseline model.
- Dropout function is necessary in the training process as it avoids the overfitting problems and help better performance on the training sets. But when the test loss is so small, adding Dropout function to decrease the test loss is limited. You can compare the method 2 with the method 3.

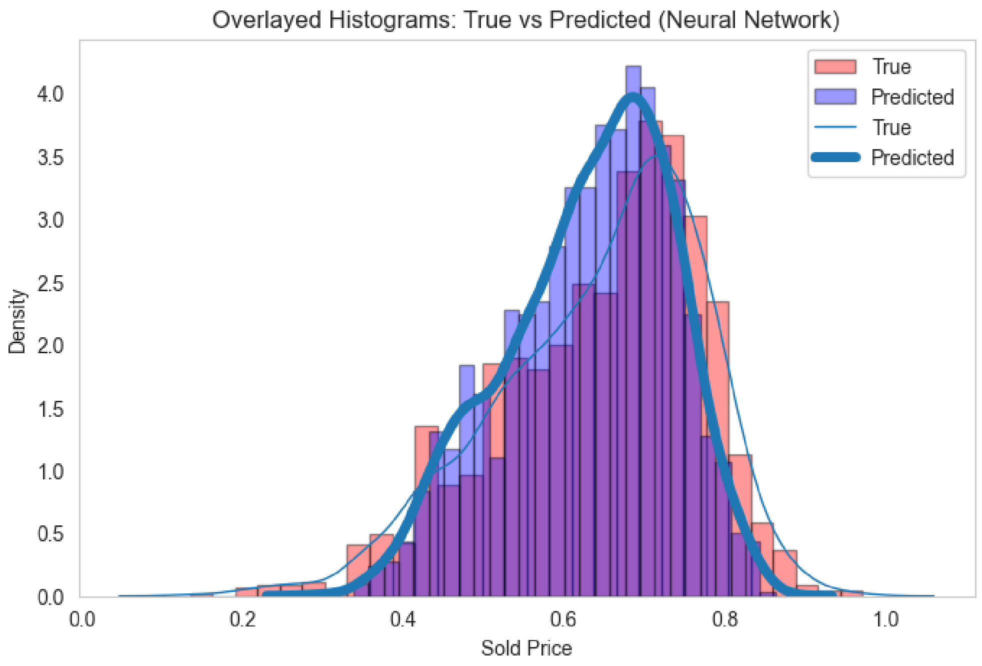

Figure 10: Neural Networks Prediction vs True

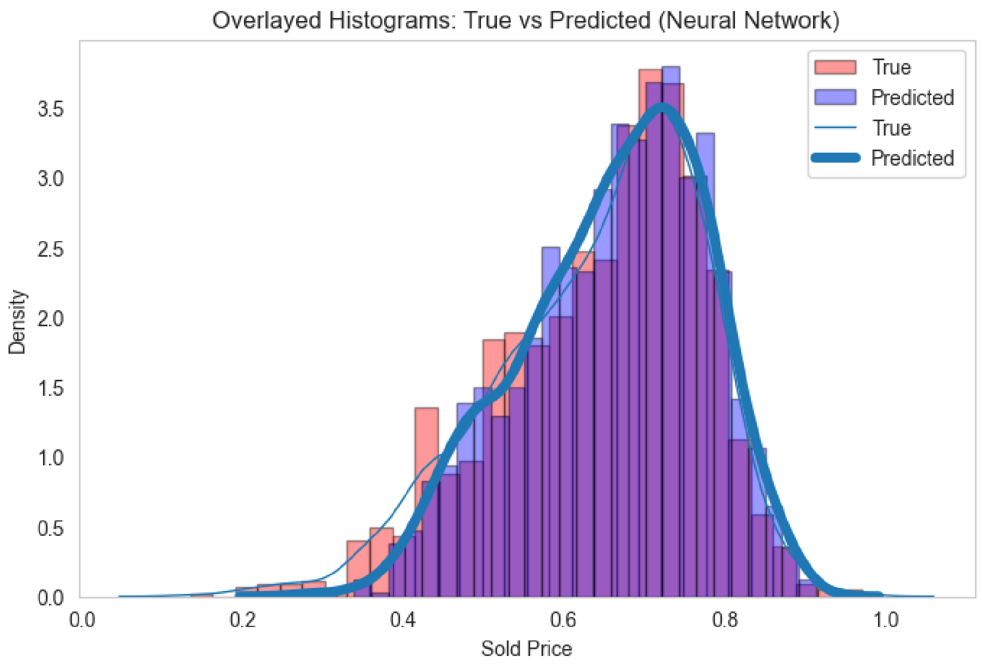

Figure 11: NN Prediction vs True (Random Seeds)

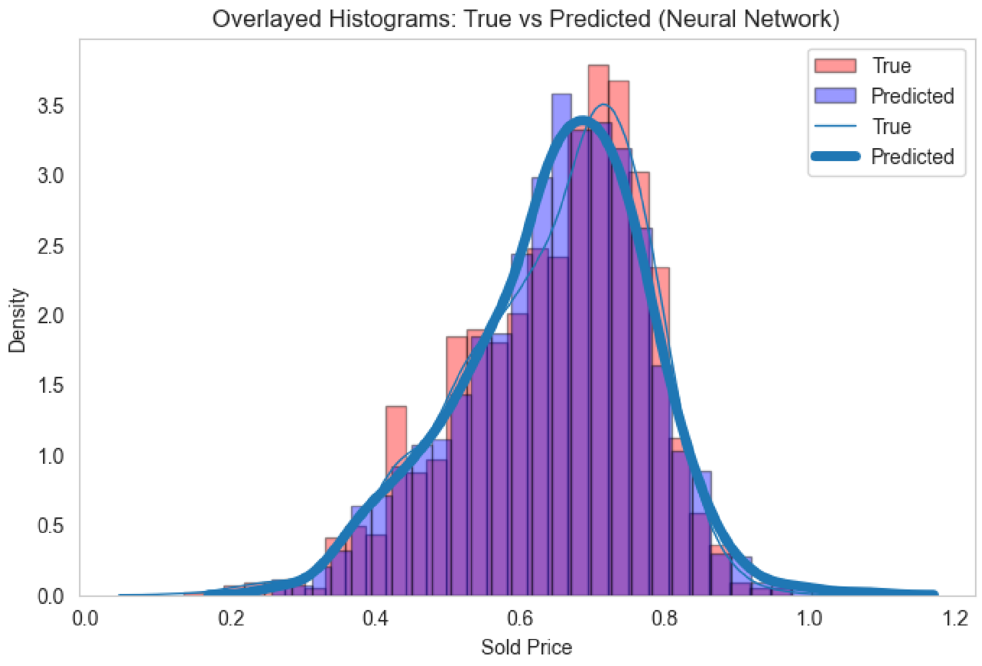

Figure 12: NN Predictions vs True (Huber Loss)

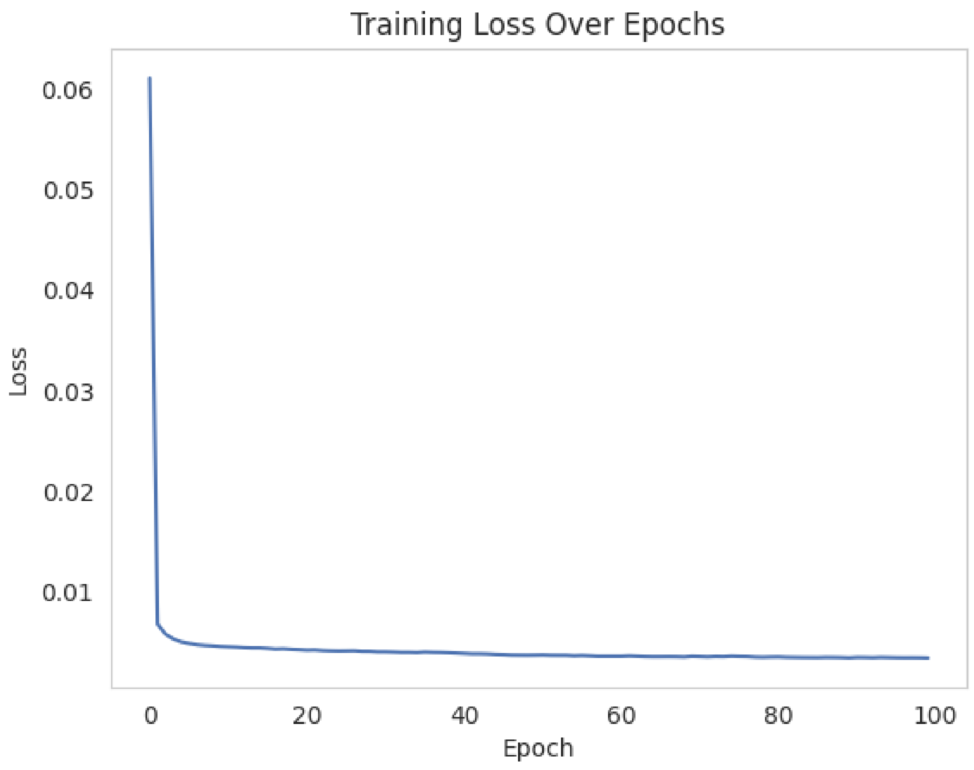

Figure 13: Loss for Method 1

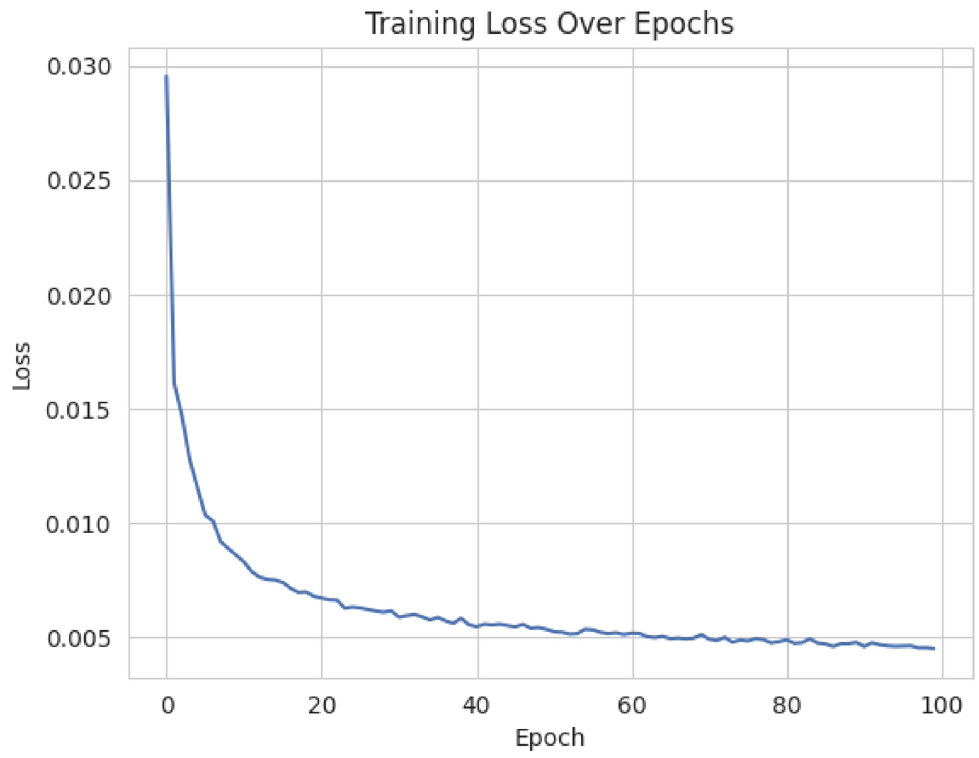

Figure 14: Loss function with Dropout(method 2)

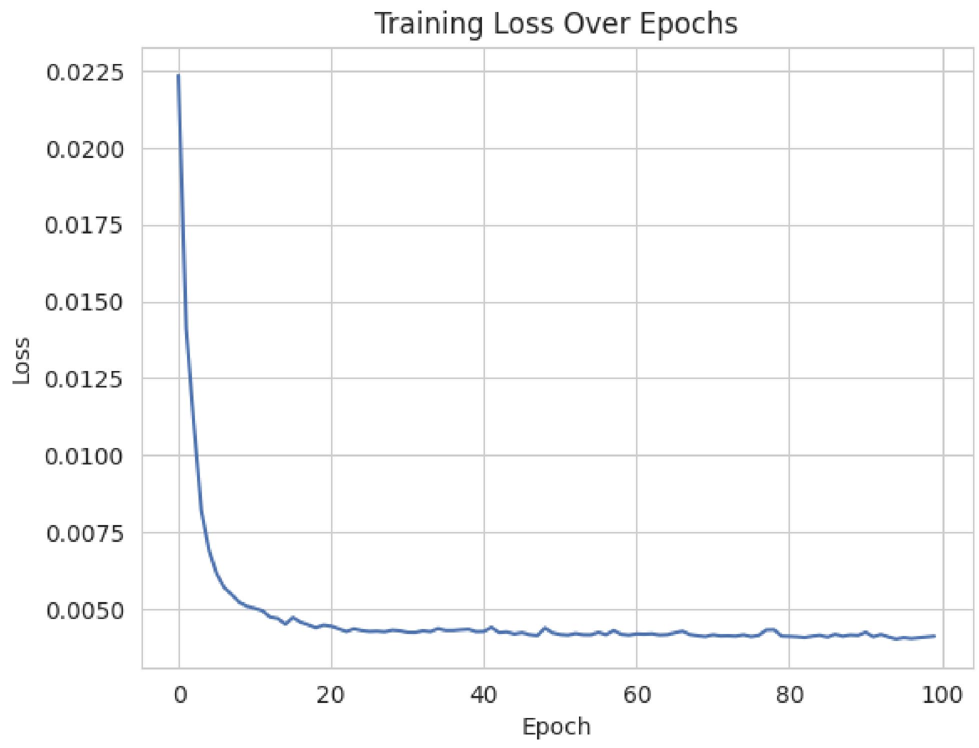

Figure 15: Loss function without Dropout(method 3)

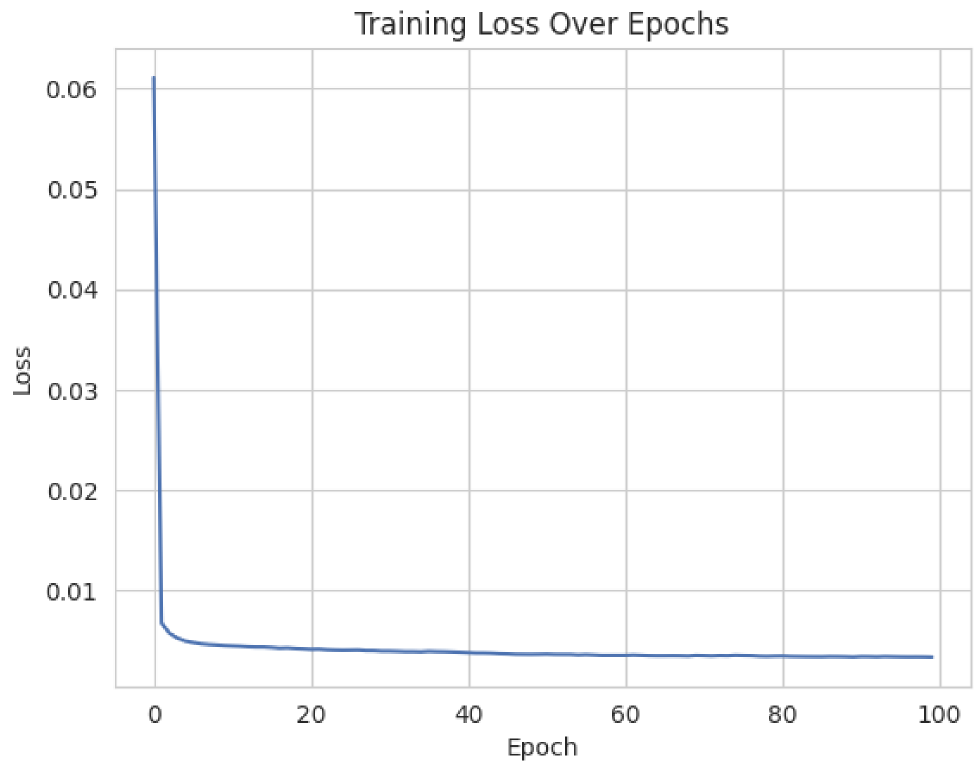

Figure 16: Loss function with ReLU activation

- Adding L2 norm and changed the optimizer sometimes don't help us improve the model performance; instead, it sometimes negatively affects the model performance. You can compare the method 2 with baseline model.
- From the MSE loss comparison, I can say that for our this model, Sigmoid function is better than ReLU function. This is likely because the target values were normalized between 0 and 1, aligning well with Sigmoid's output range. Additionally, Sigmoid provides smooth and non-zero gradients across its domain, which may have led to more stable learning. In contrast, ReLU can suffer from the 'dying ReLU' problem, where neurons become inactive and stop updating, particularly when inputs are negative or learning rates are not optimal. ReLU function can avoid varnishing gradients problems but Sigmoid can't, so I think Sigmoid might be better in this case because the vanishing gradient problem isn't severe in this model.

In conclusion, adding layers and adding dropout function increase the model performance, but its effect is limited when MSE loss is very small.

3.5 Choropleth Visualization

The Choropleth map of Illinois counties, colored by the average sale price of foreclosure properties.(Figure 17). The Choropleth map shows the predictive errors by county using the models in Neural Network method 2 as it has the lowest MSE value(Figure **??**).

The results show that our model predicts most counties accurately, but performs poorly on Marion County, where the average loss reaches 0.0099. Upon further investigation, I found that Marion appears only once in the dataset, making it extremely sparse. Since neural networks generally require a sufficient number of samples to learn meaningful patterns, the model struggles to make reliable predictions for Marion—even after standardization and adding additional layers to capture deeper relationships. This suggests that the poor performance is likely due to data sparsity, while sufficient data are easier to predict.

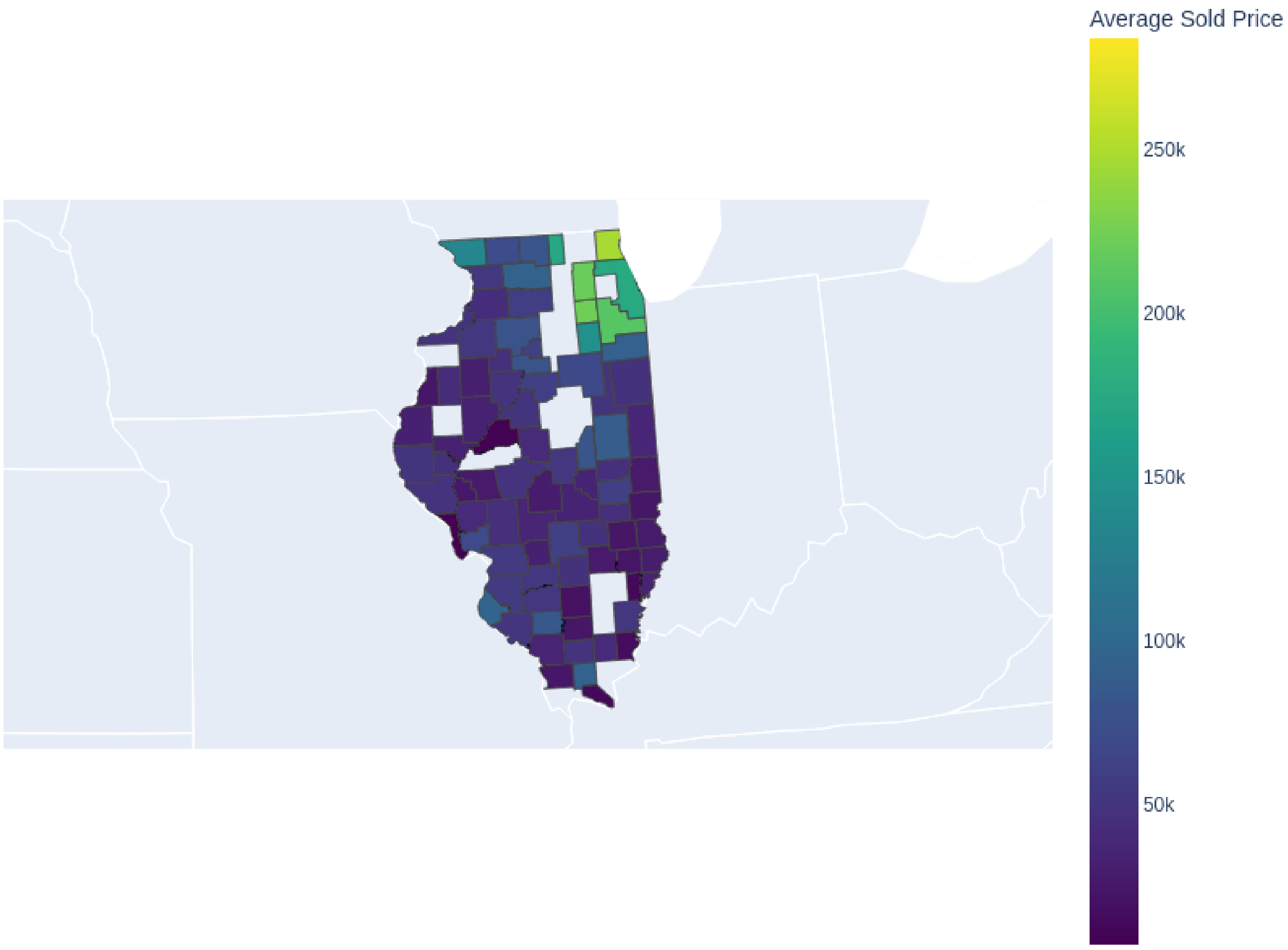

Figure 17: illinois_choropleth

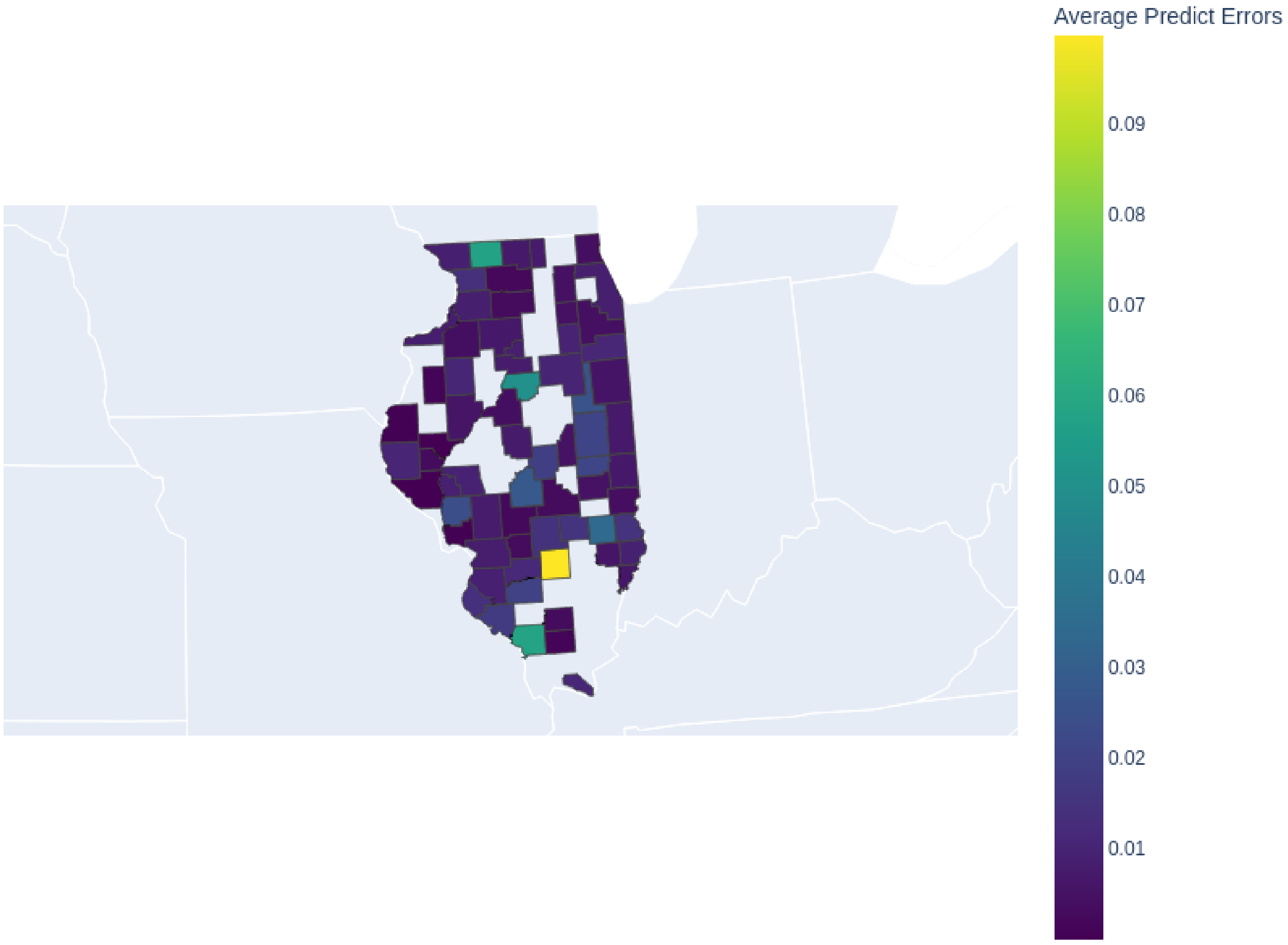

Figure 18: Predictive errors between Counties

4 Final Section-Part B

4.1 TabPFN & Linear & Random Forest Model

- **Motivation:** TabPFN has a training size limitation (typically less than 5,000 samples), so the full dataset cannot be used at once.
- **Target Binning:**
  - The continuous target variable y_{train} was discretized into 10 bins using `KBinsDiscretizer` with the `quantile` strategy as the total, since the total size of the data is 8125. But we retained its original values and not transformed into categorical bins.
  - This converted the regression task into a classification task suitable for the `TabPFNClassifier`.
- **Subset Training:**
  - The training data was split into 10 non-overlapping subsets of approximately 1,000 samples each(less than 1,000 as I round down to the integer).
  - For each subset:
    * A `TabPFNClassifier` was trained on the binned labels.
    * Predictions were made on a test subset ($X_{\text{test}}[:200]$).
    * Predicted bin labels were inverse-transformed to obtain continuous value estimates.
- **Evaluation:**
  - For each model, the Mean Squared Error (MSE) was computed between the true test values and the inverse-transformed predictions.
  - The final performance metric was calculated as the **average MSE across all 10 subsets**.

I used a new dataset, different from the previous one, where I dropped some outliers to improve accuracy. I split the target variable into 10 bins and repeated the same experiments for the Random Forest and Linear Regression models. Since these models cannot directly handle categorical features, I used the transformed dataset.

To ensure consistent scale across features, I also applied MinMaxScaler normalization for transformed dataset. For the random forest model, I used the n_estimators = 100 and max_depth = None (default). Then I computed the average MSE loss and the time we used to train these models. All are shown in the table below:

Submodel	**TabPFN**		**Linear Regression**		**Random Forest**	
	MSE	Time (s)	MSE	Time (s)	MSE	Time (s)
1	0.0139	17.14	0.0122	0.60	0.0112	1.08
2	0.0119	16.72	0.0138	0.78	0.0092	1.09
3	0.0123	16.42	0.0121	0.72	0.0106	1.01
4	0.0096	16.14	0.0128	0.66	0.0121	1.09
5	0.0118	17.01	0.0140	0.75	0.0088	1.06
6	0.0106	17.66	0.0138	0.73	0.0098	1.08
Remainder	0.0112	–	0.0389	–	0.0389	–
Average	**0.0116**	**16.85**	**0.0131**	**0.71**	**0.0118**	**1.07**

Table 5: Comparison of MSE Loss and Runtime for TabPFN, Linear Regression, and Random Forest Submodels

Comparing these three methods, we can get the following conclusions:

- TabPFN has the lower MSE than Linear model and similar MSE as the random forest model, which means TabPFN performs well in this problem. But as for the time usage, TabPFN use more than 10 times than Linear model and random forest model. So there is a trade-off here, TabPFN trades time for higher accuracy and minimal preprocessing.
- Among the three models, I recommend using TabPFN for small datasets when training time is not a concern, as it requires minimal preprocessing and only standardization to handle scale differences.

- If the model is large, I recommend using the linear regression model, because it's fast and perform quite well.
- If the model both concerns time and accuracy, I recommend using the random forest because it doesn't cost much time and the accuracy is pretty good.

4.2 TabPFN Classification

We implemented a discretization-based classification pipeline using TabPFN. The steps are as follows:

1. **Discretization:** The continuous target variable y is discretized into n bins using `KBinsDiscretizer` with the `quantile` strategy. This ensures each bin contains roughly the same number of samples. The transformed labels serve as classification targets.
2. **Subset Splitting:** Since TabPFN is designed for small datasets, we divide the training data into multiple subsets (each with 1000 samples), plus a possible remainder subset. Each subset is treated as an independent few-shot classification task.
3. **Model Training and Prediction:** For each subset:
   - A `TabPFNClassifier` is trained on the subset.
   - Predictions are made on a small fixed test set (e.g., the first 200 test samples).
   - Predicted class labels (bin indices) are inverse-transformed back to continuous values to approximate regression outputs.
   - Mean Squared Error (MSE) is computed between the **inverse-transformed predictions and the true y values**.
4. **Ensembling:** All predicted class labels across submodels are averaged and rounded to produce an ensemble prediction. These are clipped to ensure valid bin indices. The ensemble is evaluated using:
   - Classification metrics: accuracy.
   - Regression-style evaluation: MSE between inverse-transformed class predictions and true targets.

This approach enables a fair comparison between classification-based and regression-based prediction strategies under a unified framework. The results showed that the MSE value of classification methods is approximately 26.4307 and the classification accuracy is 0.3650. MSE is much higher than the regression model, so from my point of view, I think this is because some continuous values are far from the central points of the bins. And this is also the reason I vote for the regression model instead of the classification model.

Index	True Value	TabPFN	Linear	Random Forest	Disagreement Score
181	0.8281	0.8741	0.2705	0.3139	1.1639
20	0.6523	0.7664	0.2270	0.2270	1.0787
47	0.7681	0.7529	0.2270	0.2270	1.0518
109	0.6503	0.6750	0.2705	0.2270	0.8525
68	0.0882	0.2690	0.6430	0.6884	0.7934
87	0.7349	0.8337	0.4180	0.4636	0.7859
49	0.7956	0.7529	0.4539	0.2705	0.7814
115	0.5151	0.3613	0.7231	0.7263	0.7268
24	0.7448	0.8741	0.5530	0.4974	0.6978
125	0.3782	0.2690	0.6179	0.6105	0.6904

Table 6: Top 10 data points where TabPFN, Linear Regression, and Random Forest show the largest prediction disagreement.

From the table 5, we observe that TabPFN provides predictions much closer to the true values in regions where both Linear Regression and Random Forest tend to underpredict. We hypothesize the following reasons for TabPFN's better performance:

- **Bayesian Prior Knowledge:** TabPFN is pretrained on millions of synthetic tasks sampled from a broad prior distribution. This enables it to perform implicit Bayesian inference and make robust predictions even when training data is limited or noisy.

- **Few-Shot Generalization:** The model is explicitly designed to perform well on small data subsets, which helps prevent overfitting and allows it to generalize better on rare or atypical data points.
- **Robustness to Feature Anomalies:** Upon inspection of the original data, we observed that some samples contain **zero values** in critical features such as `lot_sqft`. These anomalies can disproportionately affect models like Linear Regression or Random Forest, which rely directly on the feature space. In contrast, TabPFN's learned prior and internal feature representations may make it more robust to such inconsistencies.
- **Nonlinear and Task-Aware Inductive Biases:** Unlike linear models or tree ensembles that make local decisions, TabPFN leverages a transformer-based architecture that captures global patterns and interactions across features, which may be especially advantageous when feature distributions are skewed or sparse.

In conclusion, TabPFN demonstrates superior generalization and more robust performance when the input data includes problematic feature values such as zeros or outliers.

5 Conclusion

This project analyzes foreclosure housing prices in Illinois using various models including linear regression, decision trees, random forests, and neural networks. Model performance is evaluated using MSE and distribution comparisons between predictions and true values. Regularization (L1/L2) and hyperparameter tuning are applied to improve performance. TabPFN is also tested and compared to traditional models in both continuous and discretized settings. Visualizations such as choropleth maps help identify spatial patterns and model errors across counties.

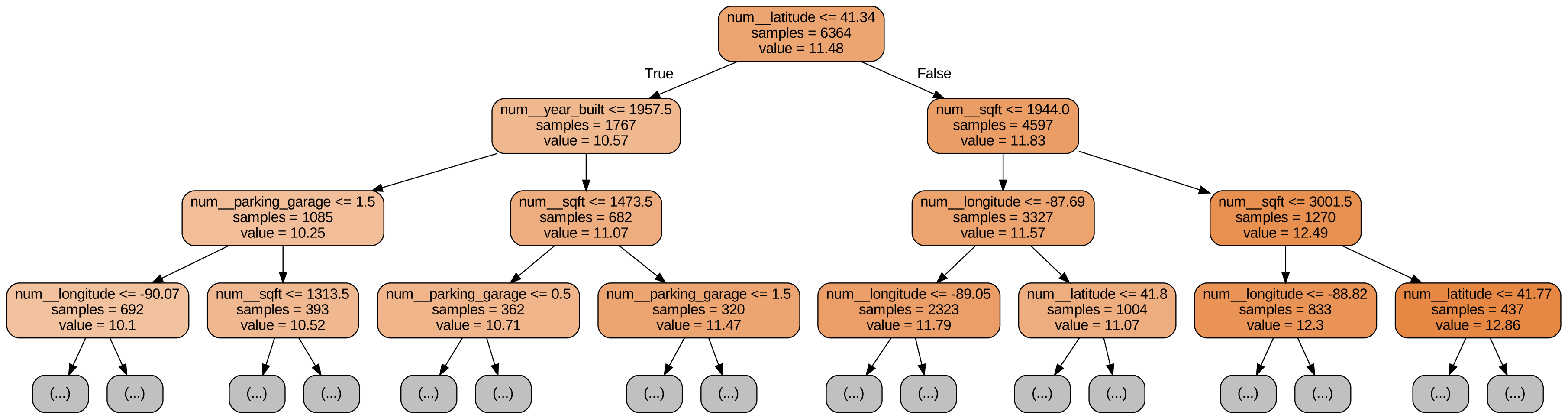

Figure 19: Decision tree visualization of the housing price model.