Housing Price Predictor
Project Deliverables

William Heinecke, Elliott Hendrickson, Daniel Lee, Yuuki Matsunari

Data Converting Process

Summary:
This document outlines the steps and methodologies and analyzes a dataset using Python libraries such as pandas and scikit-learn. The primary objective was to normalize training data and compute correlations.

Problem:
The primary challenge addressed in this project was the effective transformation and analysis of raw data to facilitate easier and more insightful statistical analysis. Specifically, the issues were:

Handling mixed data types (numeric and non-numeric) within the dataset.
Normalizing data to a common scale without distorting differences in the ranges of values.
Identifying and understanding the relationships between various features in the dataset.

Solution Overview:
The solution involved a series of data preprocessing steps implemented through Python's data manipulation and machine learning libraries. The process was as follows:

1. Data Preparation:
    a. Data was read using pandas.
    b. Irrelevant columns, such as 'id', were removed as they did not represent features of the data.
    c. Non-numeric data were identified and isolated for transformation.

2. Data Normalization:
    a. Numeric data normalization was accomplished using MinMaxScaler, ensuring all numeric values were scaled uniformly between 0 and 1.
    b. Non-numeric data were converted to numeric formats using OneHotEncoder, allowing for incorporation into the analysis.

3. Pipeline Integration:

    a. A pipeline was constructed using ColumnTransformer to streamline the conversion process from raw data to an analysis-ready format.

4. Correlation Analysis:
    a. The normalized data was subjected to a correlation analysis using pandas' .corr() function to calculate and visualize the relationships between features.

Empirical Insights and Results:
The results from the normalization process made 0-1 data ranges for each category, which allowed for more accurate correlation analysis. The correlation analysis provided valuable insights into how different features related to one another.

Regression Based Neural Network

The idea to create a neural network to solve this problem was from the kaggle website itself. Inside of the website there is a tutorial section to teach deep learning, and deep learning algorithms. One of those being a regression based neural network. Which is trained through a supervised learning algorithm, with a pre-set learning rate, batch size, dataloader, and correct values to train on. A regression based neural network, is a graph from category theory, some set of starting nodes, and some other set of ending nodes. Each individual node contains some function, so that when a value is passed to this node it performs this function and then passes its values out its paths. These paths have differing weights which affect the value passed on them. During testing once all values make it through the hidden layers to the output nodes predicted values are outputted. Although instead if the Neural Network is currently being trained, this value is compared to its true value. Which then calculates a loss using a function that changes, and then the weights, and functions change for each node regressively. Another aspect of this neural network setup that could be seen during the presentation, is how many hidden layers and nodes to use in those layers. As this is currently not solved, nor the specific functions or coefficients to use on nodes. You will see that this project is also partially an experiment, with the manipulated variable being the number of nodes, and number of layers. With the responding variable being the accuracy of the neural network.

Methodology:

Originally I did not use any manipulated data, and applied linear transformations on certain non numeric data to create the neural network. Although I later felt that this did not include my groupmates work so changed my strategy to create a more inclusive Neural Network. The next generation of Neural Networks was based strictly on data, without any other manipulation or as described in the data converting process section.

So the steps taken in order for creating each neural network was first to read the data used for that specific generation while dropping the rows with empty values. As this can and does cause errors while training the data, and creating the data loader. From here I split the data frames created from the csv, 80% and 20%. This is in order to separate the testing, and training's data. This helps prevent the neural network from cheating while testing, and gathering statistics. From here I then converted the data frames into numpy matrices, because of how they can easily be converted into a data loader. Which I did in the next step, by converting all of the row values from the input data into individual tensors, which were then combined into a data loader. A numeric tensor is necessary for Neural Networks, as the network was not created for non quantitative data. In more advanced models this can be gotten around, although for this project it is not necessary. The answers were loaded into a second section of the data loader that is only used to compare the output of the neural network. The next step I determined was the batch size, which is another thing that is not solved. So I originally tested with differing sizes, and then just made it as large as possible to fight the noise in the data. The batch size was one thousand two hundred, which worked well as I did not run out of memory during the creation of these neural networks. The next section was the creation of the Neural Network class using functions from the Pytorch library. This is the section where from generation to generation of Neural Networks created, had code changes. I first would create a hidden layer of linear nodes using a basic linearization function where I specified the number of input nodes and output nodes. The next step was to use the rectified linear unit function, which served as a basic activation function, and also removes negative relationships in the neural network. From here I would then repeat the previous steps depending on the generation with differing amounts of hidden nodes, and layers. No one in the group had a GPU, that they wanted to spare so specified that the Neural Networks, training and testing would be done on the CPU. I then specified the loss and optimizer function, both of which I determined from several online blogs for beginners. At this point we have instantiated everything required for training, testing and statistics. Which have already been explained.

Conclusion:

The differing node, and hidden layer quantities created vast differences in results. As well as the data used for the input tensors. Surprisingly, the best resulting Neural Network was one which used normalized data, but then used all of the rest of the numerical data set as from the data converting process. The second best was the last generation, which used data with only a confidence value of greater than 0.5 and also many nodes in each hidden layer. Currently there is not enough data to conclude on what the best possible neural network shape would be for this regression problem. The reason for the lack of this data is that each Neural Network would take nearly an hour to train, and as such I was limited in terms of the quantity of Networks I could create. Although as you can see for each additional hidden layer, and number of nodes the

result got better. This could either be a case of overfitting, a fluke, or during the testing the algorithm was actually approaching the optimal number of nodes and hidden layers.

Generation	Hidden Layer 1	Hidden Layer 2	Hidden Layer 3	Results %
One	64 Nodes			39.1%
Two	912 Nodes	64 Nodes		1.2%
Three	64 Nodes			10.9%
Four	128 Nodes	64 Nodes		5.2%
Five	196 Nodes	196 Nodes	64 Nodes	5.2% (Marginally Better That Fours)
Six	196 Nodes	784 Nodes	64 Nodes	2.3%

Each Generation Data Explained:

1. Non Manipulated Data
2. All Manipulated Data
3. Only Manipulated Data With A Confidence to sales price over 0.5
4. Only Manipulated Data With A Confidence to sales price over 0.5
5. Only Manipulated Data With A Confidence to sales price over 0.5
6. Only Manipulated Data With A Confidence to sales price over 0.5

Random Forest Regressor

Another method to solving our problem set was to use a random forest regressor approach. The idea for using a random forest is that you can use any combination of features, as long as they are in a numerical form. A random forest model makes use of several decision trees to try to predict the price of a house. The decision trees themselves would take in the feature set given, then calculate which feature at a certain threshold, would provide the highest variance. After the variance is calculated, the data set is split at that threshold where the new left node would contain the set that is less than or equal to the threshold, and the right side would get the set that is greater than the threshold. This process continues until either the max depth of the tree is reached, or the highest calculated variance is less than or equal to zero. The ‘random’ part of random forest refers to the feature set that is given to the trees in the forest. Before each tree in the forest is made, the data is bootstrapped so that each tree might contain a slightly different data set. The idea behind making a random forest is that on average you should be able to get a more accurate prediction from several slightly different trees. For example, if one tree gives a lower than actual prediction, hopefully another

tree will give a higher than actual prediction, then the two predictions will average close to the actual prediction.

Methodology:

The first step was to determine the features used from the dataset. To do this I looked through and chose which attributes I thought would be the most relevant to the task. I also only chose data which was in numerical form to begin with. Then I removed lines that contained null data. My goal with the methodology was to determine the optimal tree depth in the decision tree, and tree count in the random forest. To do this I would run through several different configurations of depth and tree count. For each configuration I would take the average of 50 tests on a randomized data set. When I refer to a randomized data set I mean that for each test the data was split into a training set, and a test set. The data set was randomized into the train/test split where 90% was train and 10% was for test, this way each run would contain a slightly different data set. Here are the results from my testing:

Figure 1:

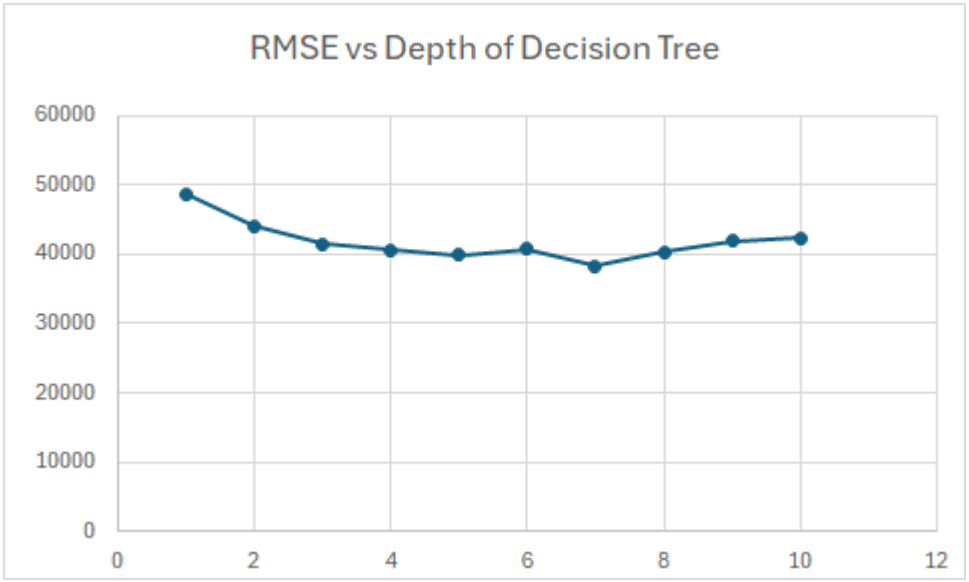


Figure 2:

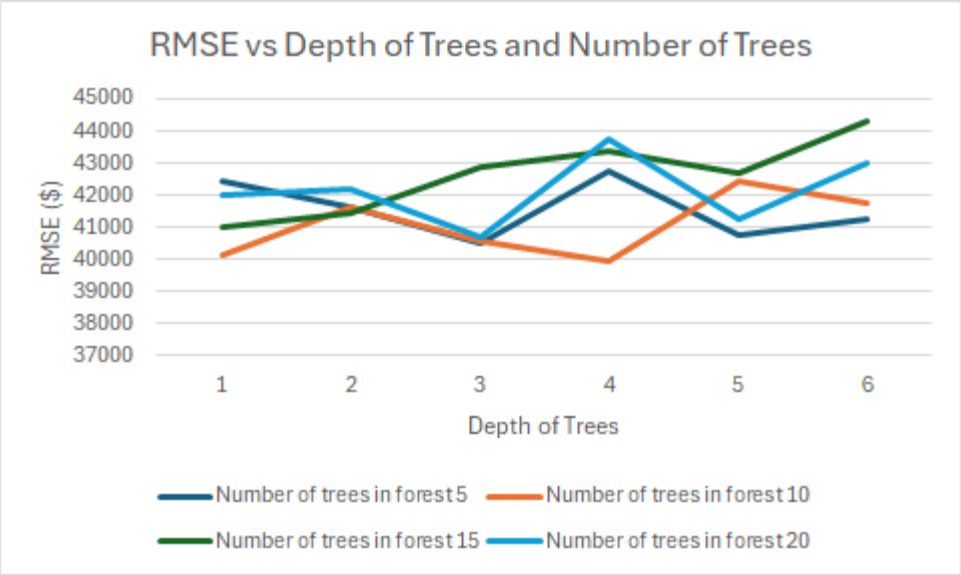


Figure 1 shows the root mean squared error (RMSE) of changing the depth of the tree while only using a single decision tree that takes all of the features as input. Figure 2 shows the RMSE of the random forests at different depths and tree counts.

Conclusion:

Overall, the random forest and decision tree models were reasonably accurate with on average only being off by 38283 from the decision tree at depth 7, and by 39933 from the random forest at 10 trees with depth 4. An interesting take away is that the random forest model is not significantly more accurate than just a decision tree, according to my results.

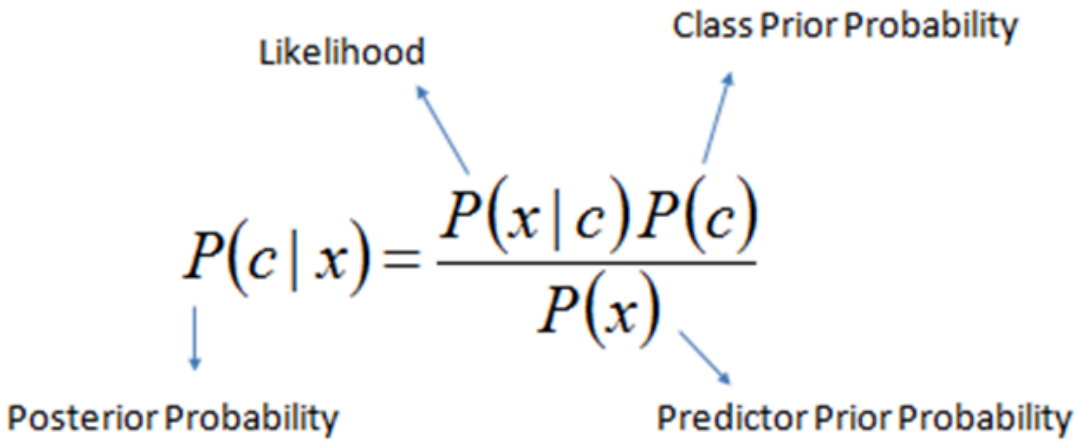
Naive Bayes

Another way to approach this problem is by using the naïve bayes algorithm to predict housing sales prices. This is a more simplified way of tackling the problem as due to the many data fields contained in our dataset that can affect sales prices, it is difficult to account for all of them while some may not be valuable at all. We instead can assume the data fields are conditionally independent of one another and more broadly predict a sales price based on a cutoff value.

Methodology:

The cutoff value is important because naïve bayes is (generally) a binary classifier for simpler cases which leads us to two major outcomes: sales prices that are >= a cutoff or < a cutoff. It needed to be approached this way since the values of predicting sales prices can vary widely, resulting in too many scenarios to predict. Our

focus is on four data fields from our dataset that initially seem valuable in predicting the sales price: SaleCondition, SaleType, YrSold, and MoSold. Through these categories, we can try to roughly predict where a sale price may lean towards based on a given estimate price (cutoff). The program follows the naive bayes formula in calculating probabilities based on a series of prior probabilities for each conditional.



$$P(c \mid \mathrm{X}) = P(x_1 \mid c) \times P(x_2 \mid c) \times \cdots \times P(x_n \mid c) \times P(c)$$

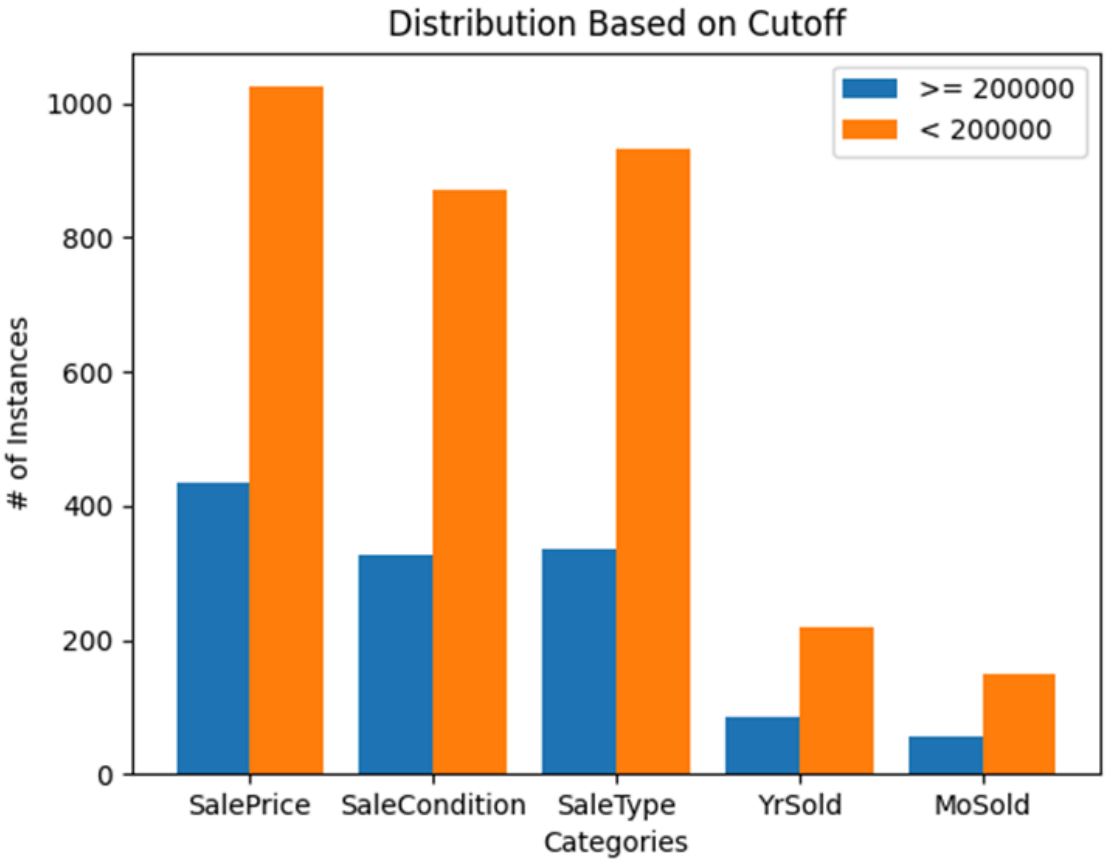
Results:

For example, if we were to choose these parameters

```
Sale price estimate: 200000
Sale Condition: Normal
Sale Type: WD
Year Sold: 2008
Month Sold: 5
```

then the outcome would be

```
>= prob: 0.0042747339809916 29
< prob: 0.016830318764406257
The sale price is predicted to be less than 200000
```



From this, we can see that based on the parameters normal, wd, 2008, and 5 that the probability that the sale price would be < 200000 is greater than the probability that the sale price would be >= 200000. This can be attributed to the fact that since there were more items that were < 200000 and matched the given input for every category, the probability values of each individual category for < 200000 would be greater than the ones for >= 200000, thus producing a higher value.

Conclusion:

The purpose of this algorithm was to try a more isolated and simpler approach at predicting housing sales prices, compared to a more thorough and complex method like a neural network to solve the problem. It can help us determine the impact that each data field has on predicting sales prices and discover trends. For the purposes of this project only four data fields were used to predict the outcome probabilities, but it could be extended to account for more, if not all the data fields provided in the dataset to better fine tune the algorithm. Different combinations of data fields could also be chosen to see how they impact sales prices.