2020

Predicting Used Car Prices with Linear Regression in Amazon SageMaker

UDACITY MACHINE LEARNING NANODEGREE CAPSTONE PROJECT REPORT

CHARLES FREDERIC ATIENZA

1 Definition

1.1 Project Overview

Supervised learning is rapidly becoming one of the most researched and utilized field of study in Machine learning in commerce. It is being used from all industries from agriculture, marketing, and even stock trading. Given the versatility of supervised learning, its application to markets outside of business-to-consumer transactions is worth exploring. Consumer-to-consumer marketplaces like Craigslist and Carousell have its own economy which is driven by the consumers themselves. Supervised learning could potentially standardize these economies so that each consumer has a sufficient understanding of what kind of factors contribute to the prices of goods on these marketplaces.

Automobiles are unlike most goods sold on consumer-to-consumer marketplaces. Its price has a lot of contributing factors. Perhaps, its most dominant factor is the car's model. Although this factor is what most buyers and sellers based their estimates from, its final price estimate further fluctuates when other secondary car specifications like mileage, transmission, drive type, and overall condition. The number of factors to consider when setting a price can be daunting for casual car owners. Supervised learning can help buyers and sellers alike on gauging a market-standard price when provided with the vehicle's many complex specifications.

Sale price estimation of goods is not uncommon in supervised learning. Most beginner-level linear regression tutorials and projects deal with the famous Boston Housing Dataset. This project aims to achieve something similar but with the challenge of a drastically different dataset structure, the Used Cars Dataset (Version 4) on Kaggle.

1.2 Problem Statement

Often on consumer-to-consumer marketplaces, sellers tend to base their items' price off other items of the same kind's price. Sellers of used cars, however, tend to come up with their own price based on estimates on various factors like the model, mileage, and condition of the car. There is also the presence of car flippers who buy used cars on below market price and sell on a considerable profit margin that does not always translate to how much the car is actually worth. With supervised learning and ample data, we can determine a reasonable price for a used car given its many quantifiable specifications.

To address this, the project aims to build a supervised model that will produce a market-level estimate of a used car's price given its many features. To ensure the best model possible is produced, three different types of model architecture will be trained and evaluated – An Amazon SageMaker LinearLearner, an Amazon SageMaker XGBoost model, and a custom PyTorch model. The models will be fed with data from the Used Cars Dataset.

Compared to the Boston Housing Dataset, the Used Cars Dataset has a more nominal columns and contains a troublesomely high number of outliers. These outliers are caused by used luxury cars on the market. These luxury cars do not necessarily decrease in value by wear and tear unlike most used cars on the market. There is also the issue of sellers who deliberately post their car with ridiculously low prices to make their post show up first on the listing page for better visibility. The project will turn these outliers to advantages by trimming out only the extreme outliers and keeping most of them in the dataset. This approach undoubtedly will affect the accuracy of the supervised models but will ultimately help the project achieve its goal.

1.3 Metrics

Given that the project's goal is prediction of a continuous value, the models will be evaluated on a metric that can communicate how far off or precise a continuous value is to another. Not only this, but the metric must be able to measure a collection of values. For this, the project will employ R-squared score for its accuracy formula. The R-squared score is the average of the distance between two values. For the models, it will represent the average of the distance between the models' predicted car price and the true car price.

$$R^2 = 1 - \frac{SS_{RES}}{SS_{TOT}} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \overline{y})^2}$$

The models' performance metrics will also include metrics on the true price to predicted price distances. Namely, minimum, maximum, and mean distance.

2 Analysis

2.1 Data Exploration

The dataset is taken from the Used Cars Dataset on Kaggle which is licensed under public domain. Its content consists of every active used vehicle posting within the United States on the Craigslist as of January 2020. The dataset contains only one file *vehicles.csv* that contains the entries.

Input data features

- *id* - entry ID
- *url* - listing URL
- *region* - Craigslist region
- *region_url* - region URL
- *price* - entry price
- *year* - entry year
- *manufacturer* - manufacturer of vehicle
- *model* - model of vehicle
- *condition* - condition of vehicle
- *cylinders* - number of cylinders

- *fuel* - fuel type
- *odometer* - miles travelled by vehicle
- *title_status* - title status of vehicle
- *transmission* - transmission of vehicle
- *vin* - vehicle identification number
- *drive* - type of drive
- *size* - size of vehicle
- *type* - generic type of vehicle
- *paint_color* - color of vehicle
- *image_url* - image URL
- *description* - listed description of vehicle
- *county* - useless column left in by mistake
- *state* - state of listing
- *lat* - latitude of listing
- *long* - longitude of listing

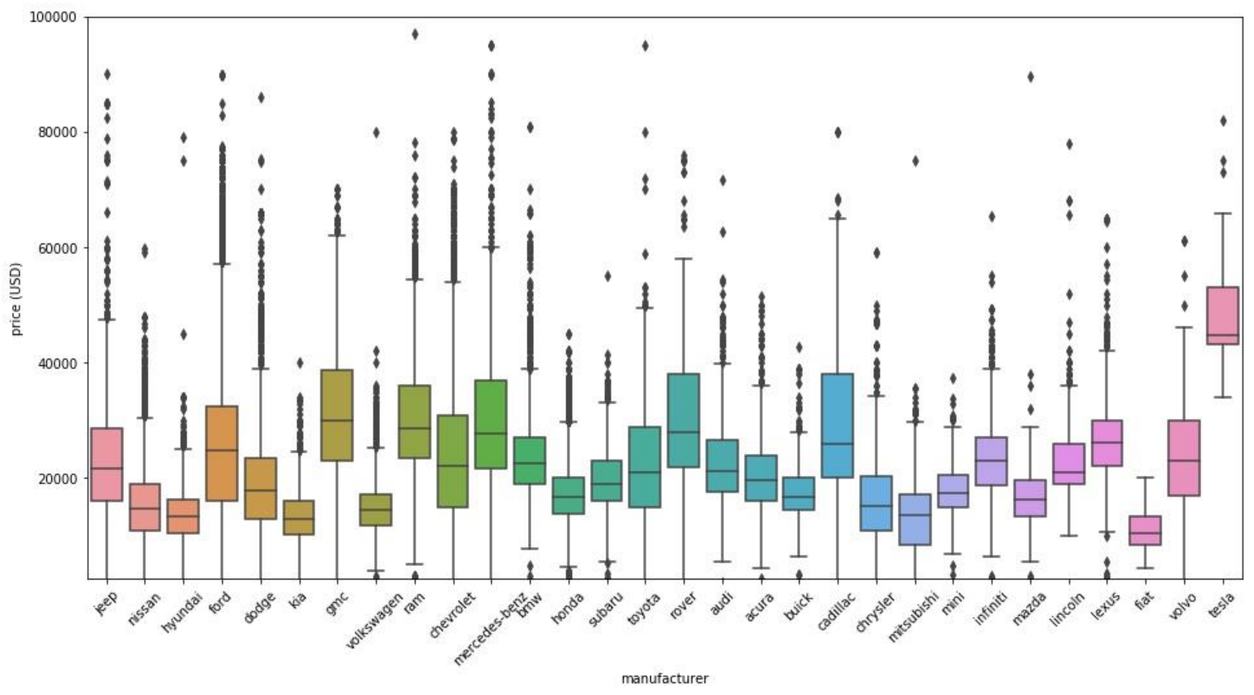
In total, the dataset has 20 columns. Most of these, like the *id, url, lat, lang*, and *description*, are Craigslist-specific features that do not have anything to do with the price so we can safely ignore them. The *cylinders* column will also be dropped. Although this column is more closely associated with car specifications than the others that were dropped, the project will be working under the knowledge that most car owners do not take the number of cylinders into account when determining a price for their car.

The project will only be exploring the columns *odometer, manufacturer, model (car model), condition, fuel, drive, size, type,* and *paint_color.* Only entries posted from 2015 to 2019 will be selected for data exploration. To avoid any confusion, the *model* feature here describes the car model. The dataset's null values will be filled with 'other' since it is what is used in the dataset for nominal features whose entry's value does not fall under the more common values for the feature. Also, 'other' for the dataset is contextually appropriate for users who opt to not specify a feature for their used car posting.

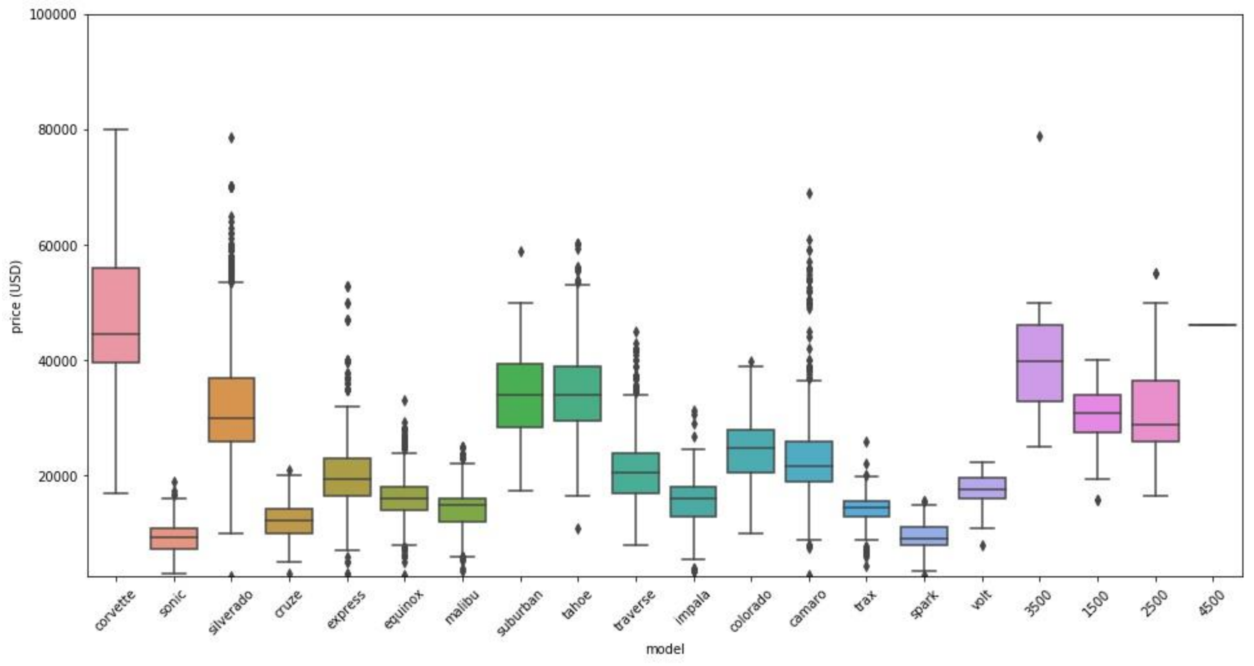
The project will limit the dataset's *price* y-axis from 2500 to 100000. The limits were decided on based on the real-world range of car prices. Although there are cars that cost way beyond USD100000, those are most likely luxury and custom cars which the project can afford to treat as extreme outliers and ignore since the goal is to build a model that accurately estimates prices of cars posted on Craigslist. The project will also treat entries whose *odometer* feature value is above 200000 as outliers and ignore them in the dataset.

2.2 Exploratory Visualization

Below is the *manufacturer-to-price* plot.

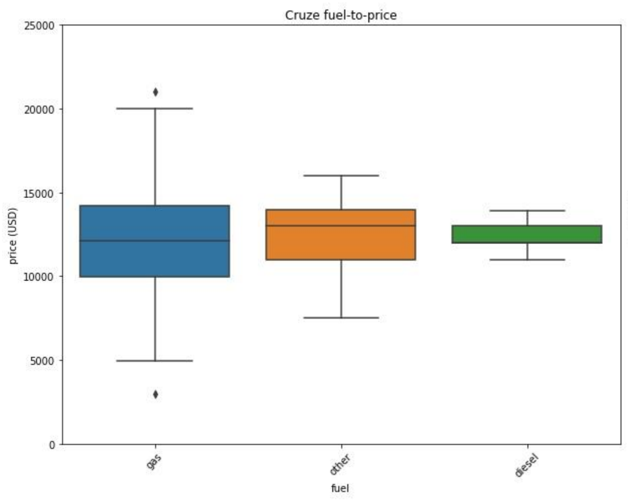


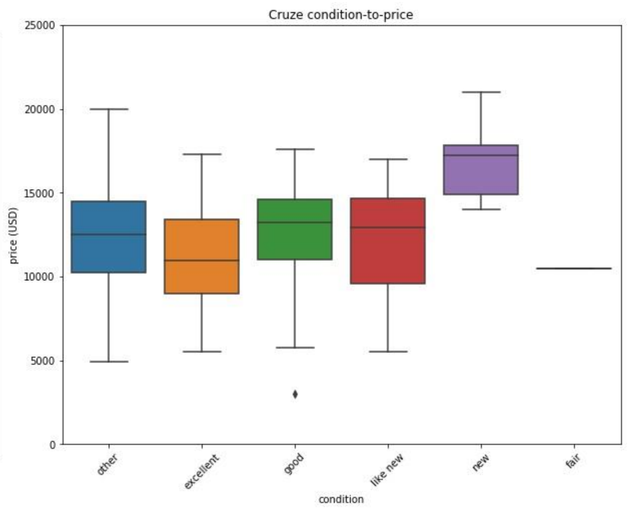
Directly plotting each feature against the *price* does not make much sense since car prices are predominantly determined by the car model before anything else. This can be observed when only car models under the **Chevrolet** manufacturer are plotted against the price.

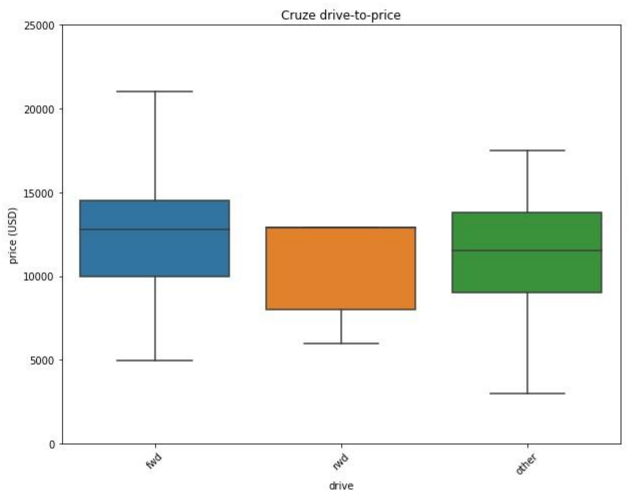


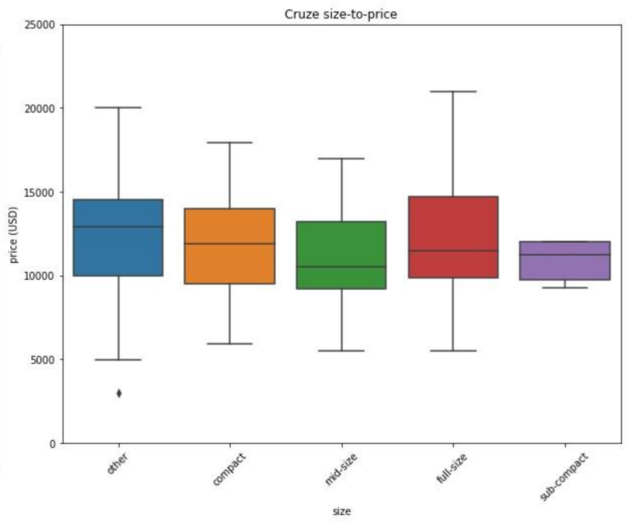
Given that the car's *model* is ultimately the biggest contributor to the price, keeping the manufacturer as a separate feature does not make much sense and doing so would only confuse the supervised models the project is building. To simplify the dataset, the *manufacturer* and *model* value will be concatenated into the *model* column then *manufacturer* column will be dropped altogether.

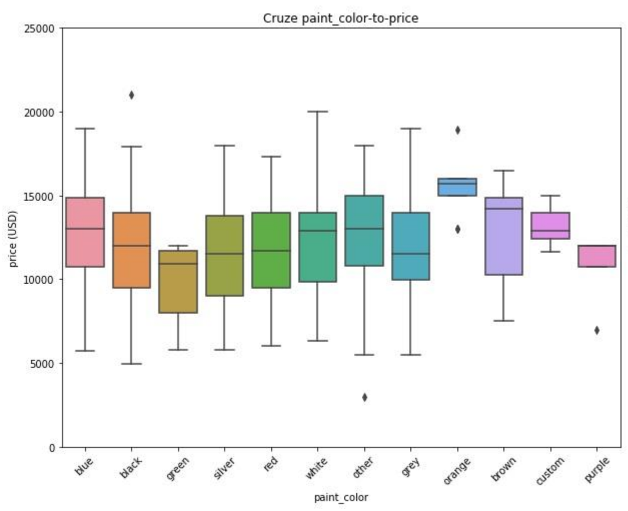
Each features' degree of correlation to the *price* varies from model to model. For this reason, it would not make sense to plot each feature to price directly. The plots will be more contextually accurate if we isolate plots by car model. The plots below are the feature-to-*price* correlation for **Chevrolet Cruze**, given by the model value *"model_chevrolet-cruze"*.

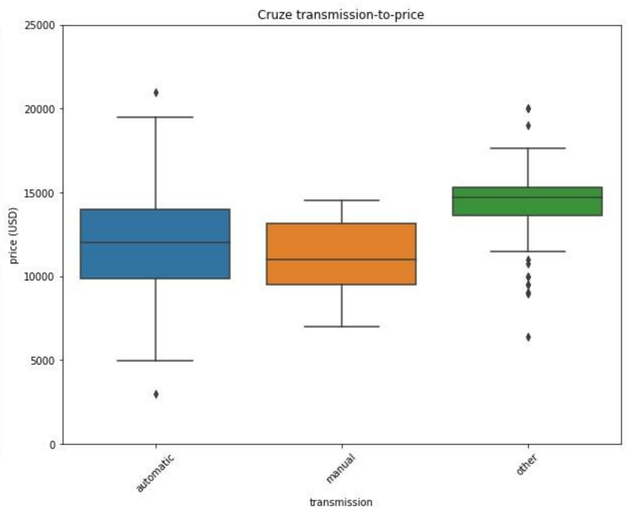


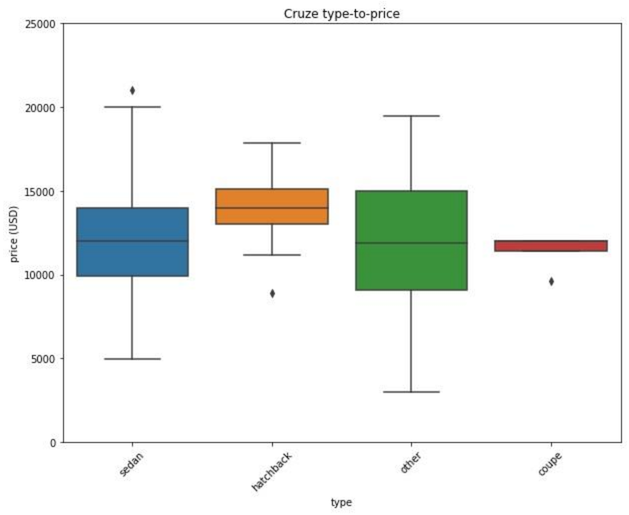




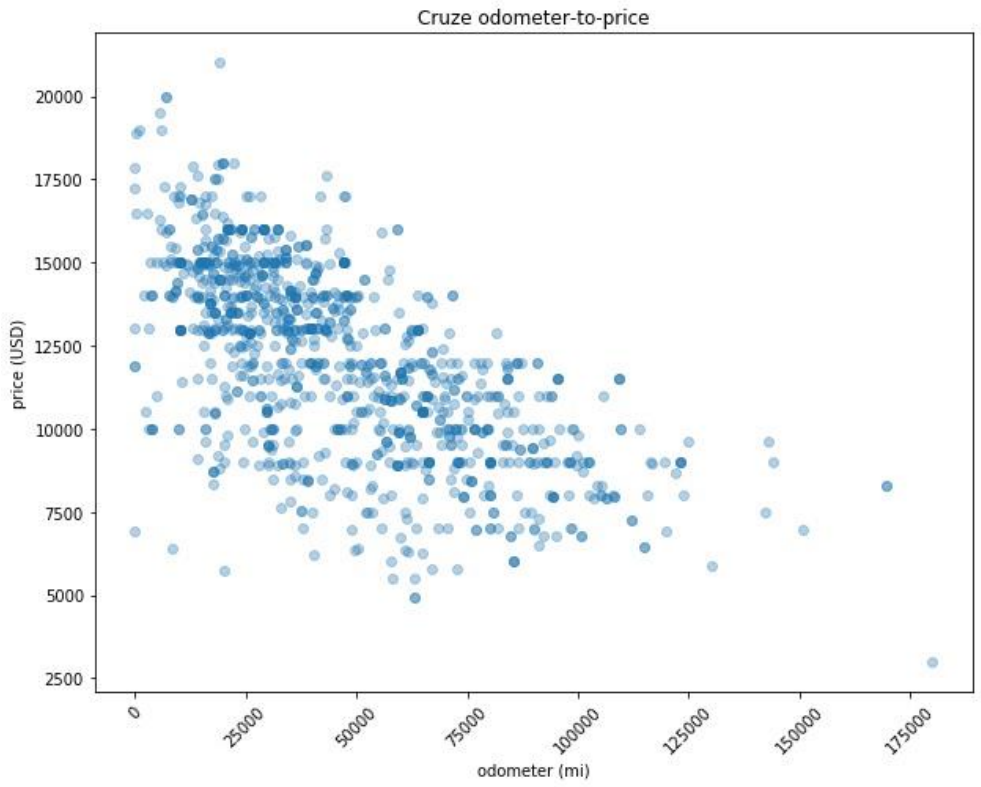








All of the above features affect the price given the distance of each feature's classes' from each other — all except the *drive* feature. The *drive* feature's class medians are closer to each other compared to the others so it is safe to assume that this feature does not affect the *price* as much and can be dropped. The dataset also contains one crucial numerical column, *odometer*, which contains the car's mileage. Even with basic domain knowledge on automobiles, it is expected that as an automobile's mileage increases, its value decreases which is evident when the *odometer-to-price* is plotted.



The project expects the *model* feature to heavily influence the models' prediction of the *price*. The car *model* distribution in the dataset is absurdly imbalanced. If our models' goal is the classification of the car *model* instead of linear regression on the car's *price,* this imbalance will prove to be a nuisance for it can make our model biased towards *model* classes of high occurrence. Nonetheless, in order to ensure our models train on a balanced range of *model*-to-*price* relations, the project will have to work around the imbalanced during preprocessing of the data with sampling techniques.

The other features' class distributions are also imbalanced. Unlike the car *model* feature, however, these barely compare in the scale of its effect on the price so this imbalance can be safely ignored.

2.3 Algorithms and Techniques

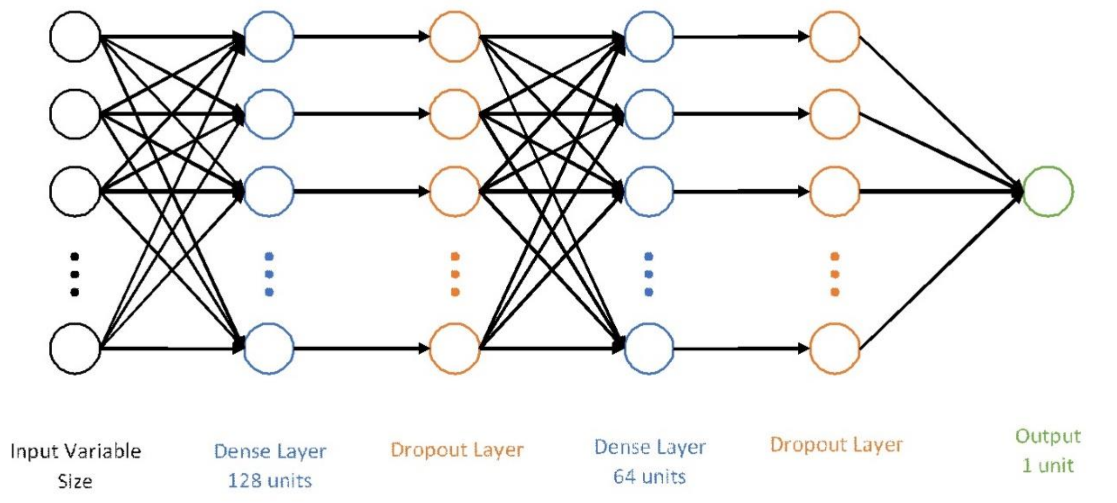
The training dataset will be oversampled and undersampled. For oversampling, the project employs SMOTE (Synthetic Minority Over-sampling Technique). SMOTE is an oversampling technique that increases the samples of the minority classes under a feature and bases the synthetic samples' features off the existing samples' neighbors. For this reason, no class may occur less than twice in the dataset. The project requires an entry to have at least five neighbors and will drop all entries whose car model occurs less than six times.

Undersampling is accomplished through the Edited Nearest Neighbor (ENN) algorithm. ENN works differently such that it removes entries whose class is of the majority in the dataset but whose other features diverge from the other entries of that class. In a sense, it will remove noise amongst the majority class entries with the goal of removing as much misclassification of class as attainable.

Both SMOTE and ENN helps the training dataset to have an approximately equal distribution of the car *model* feature which is ideal for our models to generalize and avoid bias towards the raw dataset's imbalanced car *model* distribution.

The three models the project will be training and evaluating are an Amazon SageMaker LinearLearner model, an Amazon SageMaker XGBoost model, and a custom PyTorch model. LinearLearner and XGBoost are built-in models provided by Amazon SageMaker. Both support binary classification, multiclass classification, and linear regression. The project will set these algorithm's objective to linear regression given that the goal is to estimate the car's *price,* a continuous value.

The custom PyTorch model consists of two fully-connected hidden layers with ReLu activations. There is also a dropout layer between the second hidden layer and the output layer. Since the project expects a continuous numerical value as output, the final layer will also be a fully-connected layer with one neuron.



The LinearLearner and XGBoost models automatically normalizes training input and denormalizes its prediction during inference. To have the same behavior with the custom PyTorch model, it will utilize min-max scaling to normalize its inputs and denormalize its outputs. Min-max scaling scales the values in each column from zero to one based on the value's ratio to the minimum and maximum value of the column it is in.

2.4 BENCHMARK

The models trained in the project will primarily be evaluated by their R-squared score compared against one another. The R-squared score is the best indicator for the models' performance because it measures the collective accuracy, with respect to distance, for the whole test dataset. Although in extreme cases of bad performance R-squared can go below zero, it only can go as high as one which essentially means a model accurately predicted all cars' prices on the dataset with exact precision.

The LinearLearner and PyTorch model's training and validation loss function is Mean Squared Error (MSE) while the XGBoost model's is Root Mean Squared Error (RMSE). These two loss functions are best used for linear regression because it tells the degree of how far off a model's predictions are as oppose to models with classification objective where the loss function's concern is just whether a prediction is correct or wrong.

Other metrics on which the three models will be evaluated are minimum distance, maximum distance, and mean distance. Minimum distance is the smallest prediction-to-label absolute difference while maximum distance is the largest. Mean distance is mean across the prediction-to-label absolute difference. These metrics communicate the best and worst cases for the models' utilization. The maximum distance, in particular, can be used to determine whether the models actually achieved the project's objective. A large maximum distance can indicate that the models actually estimated an outlier in the test dataset's market-average price as oppose to its label in the dataset which is either inflated or way below the real-world market prices.

3 METHODOLOGY

3.1 DATA PREPROCESSING

The *cylinders* column will be dropped for the reason stated before – car owners are less likely to factor in cylinder count when coming up with a price for their used car. The *drive* column is also dropped since its classes' median *price* distribution are too close to each other to make a difference. Of the twenty columns, only *manufacturer, model, odometer, condition, fuel, size, type, and paint_color* will be taken for data preprocessing*.*

The model feature requires additional clean-up. This is because unlike other features, the *model* column's exact text value can vary greatly depending on the poster's prerogative. For example, a car's actual model can be '**Cruze**' but the poster might opt to put in '**Cruze 2017 AT**' instead which is essentially the same category but, if left unhandled, will result in its separate category. This can be seen by getting the occurrence count of each class in the *model* column.

```
f-150                        3128
1500                         2548
silverado 1500               2201
2500                         1355
altima                       1351
                             ...
rogue sv automatic              1
escalade premium awd            1
transit-250 cutaway             1
tlx v6 w/tech                   1
patriot sport 4x4 suv           1
Name: model, Length: 8897, dtype: int64
```

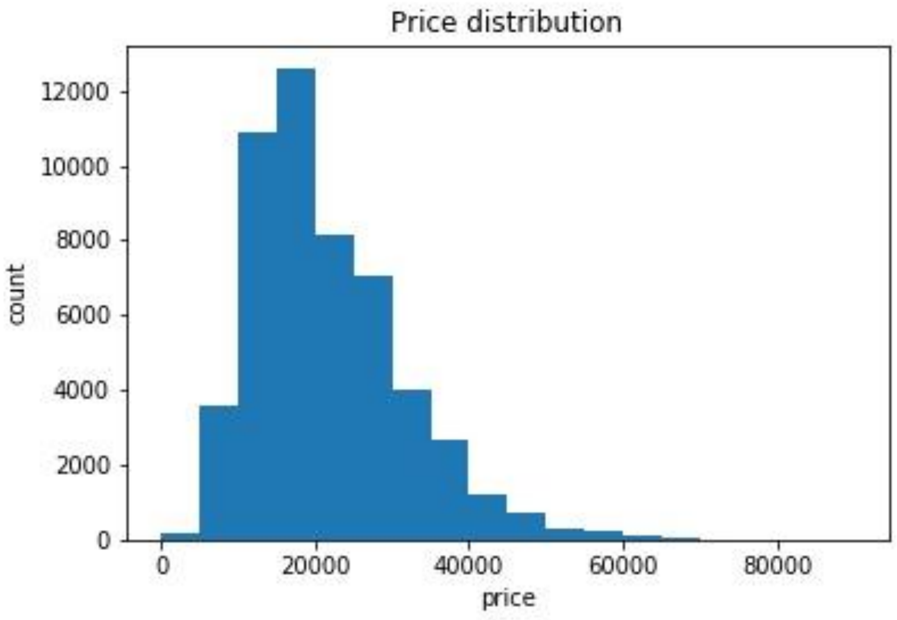
To remedy this, containment will be used to normalize redundant *model* classes. So '**Cruze 2017 AT**' will simply be normalized to '**Cruze**'. If not done, the supervised models will treat the two values as separate values which will likely lead to it not training as well due to an extremely high number of unique values of the *model* feature. To further simplify the dataset and trim out outliers, the dataset will be dropping rows whose *model* class occurs in the dataset less than fifty times.

The *manufacturer* column will be concatenated into the *model* column and then dropped altogether. The reason being *manufacturer* is redundant with the *model* column which is the heaviest contributor to the car's *price* and keeping the *manufacturer* column will only contribute to the noise and confuse the models the project is training.

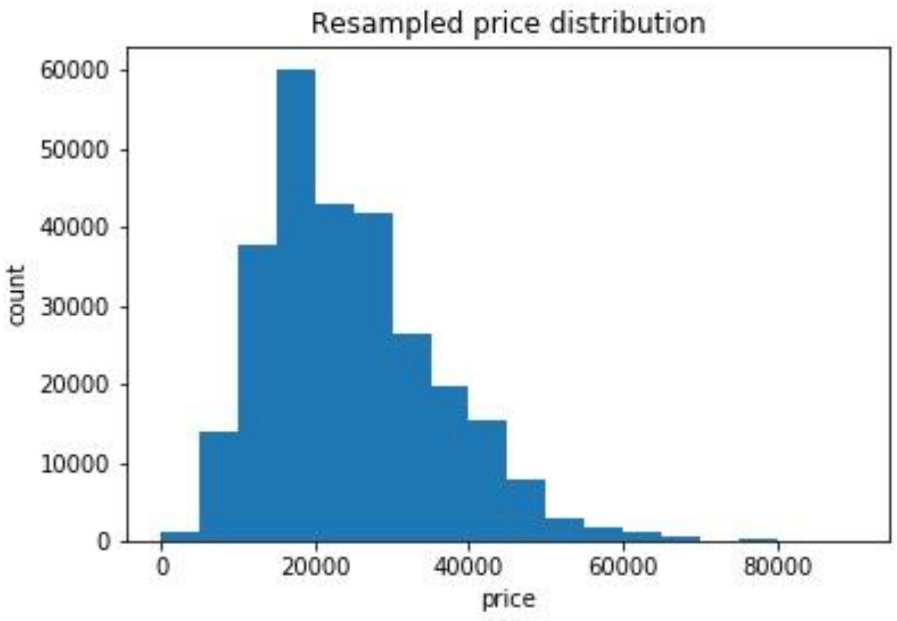
For models to properly process the dataset's numerous nominal features, they have to be one-hot encoded first. Out of the nine columns left, seven are nominal and only *price* and *odometer* are numerical. After one-hot encoding the seven nominal columns, the original seven columns will be dropped except the *model* column which will be used as the target for sampling the data. The dataset is now left with 103670 rows and 295 columns. From this dataset, 50% will be taken for the training set, 25% for the validation set, and the remaining 25% for the test set which will leave datasets with the below dimensions.

- Training set — 51835 rows x 295 columns
- Validation set — 25918 rows x 295 columns
- Test set — 25918 rows x 295 columns

The extreme imbalance of the car *model* class may cause the models to be biased towards higher occurring classes. The raw test set's price distribution is highly skewed and imbalanced.



To remedy this, the training set can be sampled where the minority classes are oversampled and the majority classes are undersampled. The project will use the *imbalanced-learn* Python package which has implemented SMOTE and ENN in a combined function, *SMOTEENN.* After running the function on the training dataset, it produces a resampled dataset with 274019 rows and 295 columns and the below price distribution.



After sampling, the *price* distribution plot is still skewed but with a slightly longer tail. But what is important here is that the distribution of price is not so far from each other anymore. Taking a look at the y-axis labels of the graph, it can be observed that the higher occurrence car models have been drastically lessened and the lesser occurrence car models increased.

The car *model* feature can now be dropped from the three datasets, note that this feature has already been one-hot encoded earlier. This will leave the datasets with 294 columns.

3.2 Implementation

3.2.1 Accuracy Metric

Originally, the formula for the accuracy metric was the one given below. It proved to not be able prediction-to-label differences where the difference is greater than the label. It produced unexpectedly large negative values for accuracy when taking the mean of the accuracy of the predictions.

$$accuracy = \frac{\sum_{i=1}^{n} 1 - \frac{|x_i - y_i|}{y_i}}{n}$$

R-squared does not have the same limitation. Because it takes the square of the differences, it can handle huge differences and still produce a value that communicates how well the models are performing.

3.2.2 *Model* Column Normalization

The *model* column normalization with containment is a special function that takes *model* column values and simplifies the longer value to the shorter one if the values have a containment score of at least the length of words in the short value over the length of words in the longer value.

3.2.3 Linear Learner Model

The first model will train and test a model using one of Amazon SageMaker's built-in algorithms, Linear Learner. The Linear Learner algorithm is an easy-to-use and straightforward model. It can handle binary classification, multiclass classification, and linear regression. Since the desired output is a continuous value, the project will make use of the algorithm's linear regression.

For the train and validation dataset, they were arranged so that the first column is the data label — the *price* feature. This is a requirement by SageMaker's built-in algorithms when feeding it data in CSV format, however, for this model, data will be fed in through the *RecordSet* object which requires the features and labels to be supplied separately.

```
X_train = train_data[:,1:]
y_train = train_data[:,0]

X_validation = validation_data[:,1:]
y_validation = validation_data[:,0]

# LinearLearner estimator object instantiation code here

X_train_np = X_train.astype('float32')
y_train_np = y_train.astype('float32')

formatted_train_data = linear.record_set(X_train_np,
labels=y_train_np)

X_validation_np = X_validation.astype('float32')
y_validation_np = y_validation.astype('float32')

formatted_validation_data = linear.record_set(X_validation_np,
labels=y_validation_np, channel='validation')
```

The Linear Learner estimator is then trained by supplying the training and validation RecordSet objects.

At the end of training, the logs produce a ridiculously high number for a validation loss — 30230839.25. This loss value is this high because, unlike the training dataset, Linear Learner does not normalize the labels of the validation set. The dataset's labels or car prices are usually in the 4000 to 12000 range so it is no surprise that the training arrived at such a value for the validation loss.

This, however, does not mean that the model is not learning at all. What we should be paying attention to is how much the validation loss has decreased throughout the training process. It has gone from 31856580.89 to 30230839.25 in 16 epochs.

3.2.4 XGBoost Model

The second model, XGBoost, is another of Amazon SageMaker's built-in algorithms. XGBoost has a more advanced architecture and is generally a more complicated function. Full understanding of XGBoost's tree-based approach is beyond the scope of the project. It does support linear regression which is why the project opted to use it here.

For the XGBoost model, dataset will be fed to through CSV files so no further manipulation of the data is needed since it already has the labels(price) as its first column. However, these CSV files need to be uploaded to the SageMaker session's S3 bucket first.

After instantiating the XGBoost estimator, the project will employ Amazon SageMaker's Hyperparameter Tuning to get the best model possible. Essentially, Hyperparameter Tuning trains multiple models with different hyperparameters after which the best performing model will be taken and attached to an estimator. In the project's case, total of twenty models was trained.

Like the LinearLearner model, the XGBoost model does not normalize the labels so it ends up with very high losses as well given the range of the car prices. Although they are exponentially lower here because the model uses Root Mean Square Error (RMSE) for validation loss, again, what is important is that the validation loss decreases throughout the training process. In this instance, the validation loss decreased from 19140.1 to 4988.79 by the end of training.

3.2.5 PyTorch Model

The third and final model is a custom PyTorch model with the architecture and output mentioned in the Algorithms and Techniques section. Since the project is working with a custom model, the training function had to be implemented. It supports early stopping in case the validation loss does not improve after a certain number of epochs and stop training early.

Before the datasets are fed to our model, its features will have to be normalized first. The project used *scikit-learn's MinMaxScaler* for min-max scaling normalization. After normalization, the train and validation dataset are put into a *DataLoader* which is the object the train function expects the dataset to be supplied in.

For the optimizer and loss function, the project used Adam and Mean Squared Error(MSE) respectively. The model will train for a maximum of 300 epochs.

```
Epoch: 1, Train loss: 843259303.4766355, Validation loss:
556914428.3076923
...
Epoch: 203, Train loss: 21587198.14953271, Validation loss:
26712609.230769232
Early stopping condition reached. Stopping training
```

A similar scale can be observed for the validation loss compared to the earlier models. Going through the logs, it can be seen that the validation loss decreased from 556914428.31 to 26712609.23 before early stopping kicked in.

3.3 Refinement

The initial approach for the custom PyTorch model was to feed the dataset to the model directly without any normalization. This caused the gradients to explode out of proportion. After only the first epoch, the loss value produced by the loss function reached positive infinity and after a few more epochs, the values became *NaN*. This can be remedied either by gradient clipping or features normalization. The project went with features normalization.

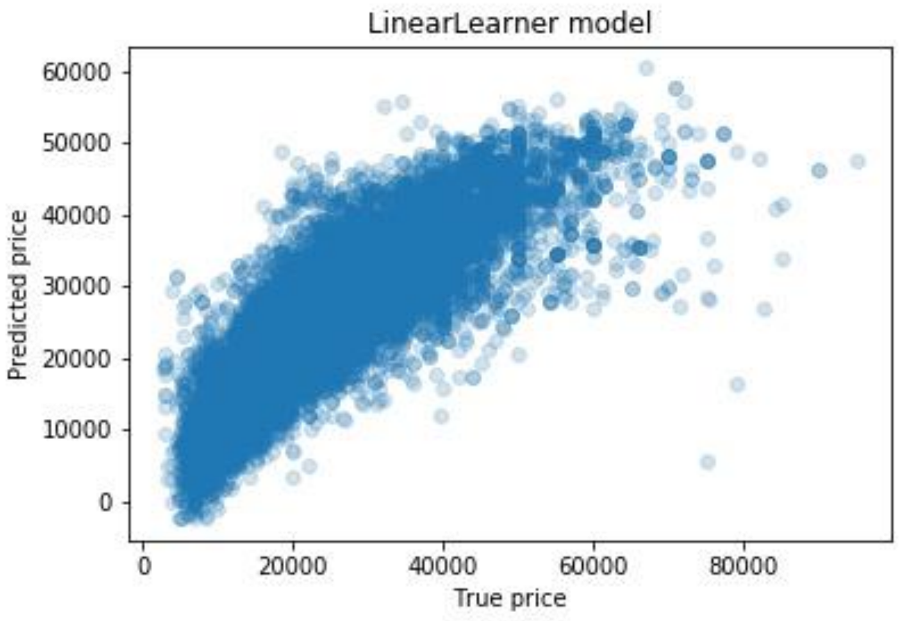
The project also initially normalized even the dataset labels when training and evaluating the custom PyTorch model. Although the validation loss still decreased, the values were too close to zero that their scale became difficult to interpret compared to the LinearLearner and XGBoost validation losses. For this reason, the project kept the label scales when feeding data to the PyTorch model.

4 Results

4.1 Model Evaluation and Validation

4.1.1 Linear Learner Model

In order to make predictions from our trained model, it needs to be deployed as an endpoint first. The test data is then loaded and supplied to the endpoint for inference. The Linear Learner model's predicted prices compared to the true prices can be plotted. The closer the plot is to a y = x function, the better.



As can be observed, the plot is more bent than straight implying that the Linear Learner model is overestimating the prices as the true price label increases.

To represent how well the Linear Learner model performs numerically, the project employed the R-squared algorithm.

```
R-squared score: 0.7372068518275521
```

When ran through R-squared, the model's accuracy score on the test dataset turns out to be 0.74. The R-squared score can become negative but can only go as high as 1.0. Given that the score is over the 0.50 mark, it is safe to assume that the Linear Learner model performed fairly. Below are the other metrics of our model's predictions.

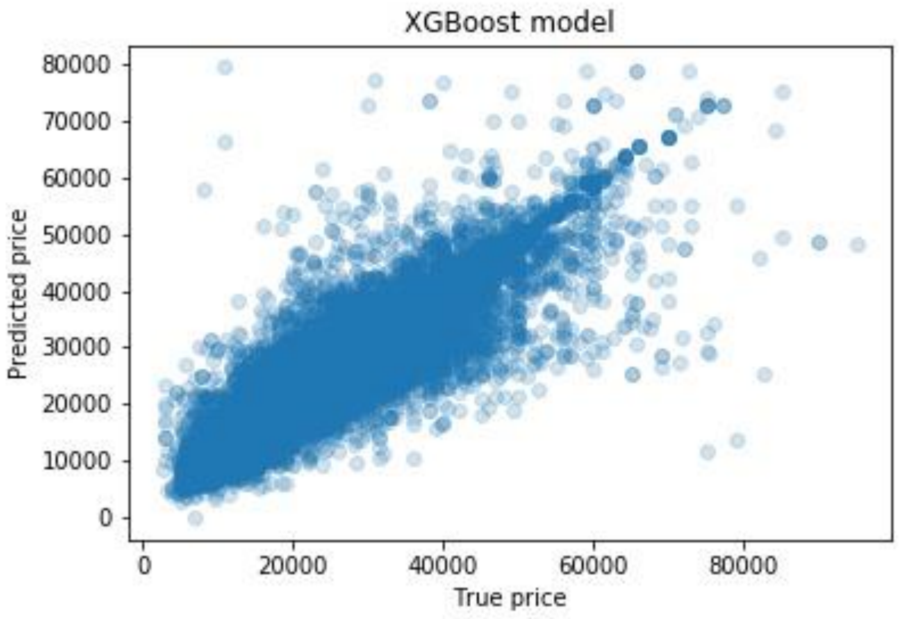
```
Min distance: 0.3203125
Max distance: 69425.931640625
Mean distance: 3898.957455078954
```

Looking at the metrics, what stands out is the abnormally high max distance value. This value, and why it is actually a good thing when the project's objective is kept in mind, is explored in the Conclusion section.

4.2 XGBoost Model

To run predictions on the XGBoost model, the project created a transformer job and supplied it with the location of the test dataset. A transformer job uses SageMaker's Batch Transform feature to conduct inference on large datasets.

When the transform job finished, the output file from the transform job's output path is downloaded and compared against the true labels.



Going by the plot, it can be observed that the XGBoost model did significantly better than the LinearLearner model. Its plot, although has more noise, is a lot straighter meaning the predicted price and true price are closer to each other.

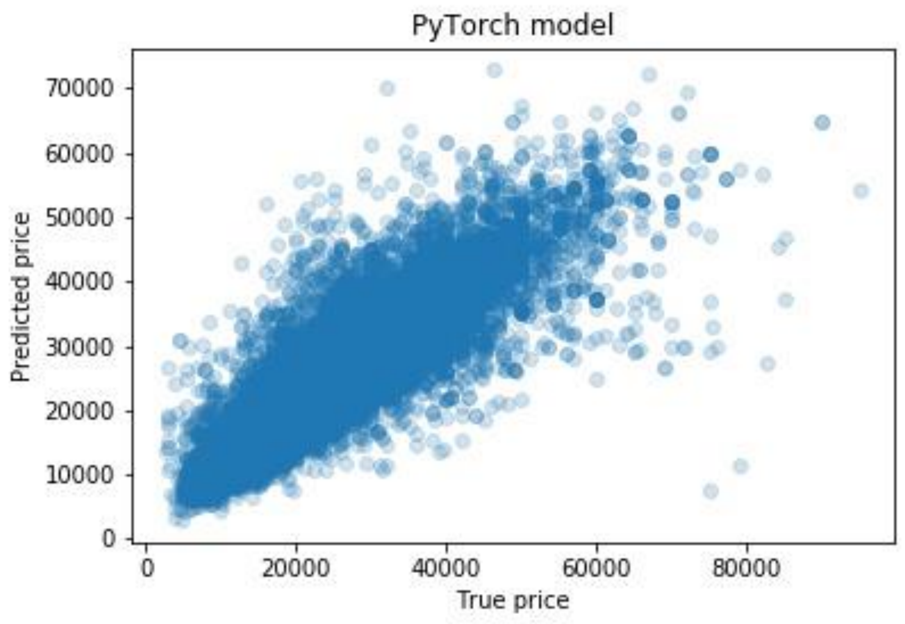
```
R-squared score: 0.7832844072803054
```

The XGBoost model also performed significantly better on our accuracy metric with a 0.78 score compared to the Linear Learner model's 0.73. On some metrics, however, it performed mostly the same except for a lower mean distance.

```
Min distance: 0.06835940000019036
Max distance: 68591.40625
Mean distance: 3193.5946961869945
```

4.3 PyTorch Model

Before inference, the test dataset's features will have to be normalized first before inference in order to get predictions on a similar scale with the train dataset. It is important that the test dataset is normalized with the same scaler that was used to normalize the training dataset to ensure that both datasets are scaled equally.



From the plot, it can be inferred that PyTorch model performed similarly to the XGBoost model — a mostly straight plot but with some outliers.

```
R-squared score: 0.7699063746869551
```

The PyTorch model's accuracy is a little below the XGBoost model at 0.77. Along with this, other metrics below are also very close to that of the XGBoost model.

```
Min distance: 0.056640625
Max distance: 67580.33642578125
Mean distance: 3468.076180230625
```

4.4 Justification

Based off the accuracy metric, the XGBoost performs the best with 0.78 R-squared score narrowling beating out the custom PyTorch model's 0.77. But all three models generally performed fairly except for an alarmingly consistent high maximum distance value. This high value is also most likely what is causing our accuracy scores to not be in the .80-.90 range.

```
Max distance car true price: 75000.0
```

Working off the PyTorch model results, the actual car price of the car that has the max distance from the predicted price is 75000. The actual car's model is needed to put this price in perspective.

```
Max distance car model: model_hyundai-elantra
```

After finding out that the car is a Hyundai Elantra, a quick search on how much it is on the US used cars market yields that its price ranges from 5000–11000 USD, way beyond its label price in our test dataset. The high price can also be put in a context in terms of test dataset by getting the mean price of all Hyundai Elantra cars there are in the test dataset.

```
Max distance car model mean price: 12317.555970149253
```

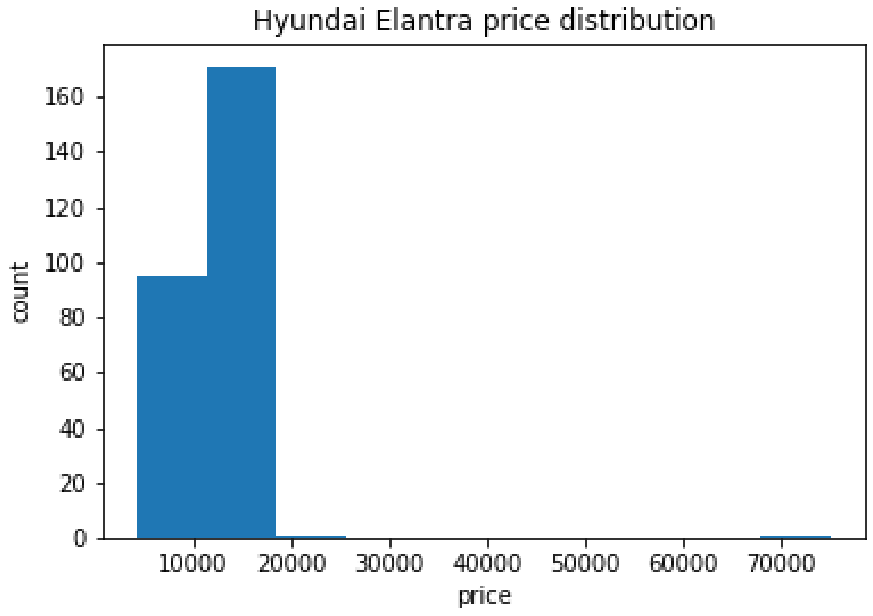
The mean is 12000 USD. This, again, is way below the label. Form all this, it can be concluded that this is an outlier in the dataset.

This proves that the models work. The objective of the project is to build a model that can estimate a market-average price for used cars in order to help buyers and sellers avoid inflated car prices on the consumer-to-consumer market. When someone heads to Craigslist looking to buy a used Hyundai Elantra priced at 75000 USD, running its specs on one of the models will make him realize that this particular Hyundai Elantra's price is well above market and he should look somewhere else.

5 Conclusion

5.1 Free-Form Visualization

The project's goal is to build a supervised model that can estimate a fair, market-standard price for used cars. It is evident that, at least for the test dataset, that the models built achieved this.



Going by the price distribution of Hyundai Elantra cars in our test dataset, the entry that had the max prediction-to-label distance is evidently an outlier in the dataset seeing as it appears to the very edge of the plot. Real-world and test dataset prices for Hyundai Elantra cars range from USD 5000 to a maximum of USD 20000. The extreme case of a Hyundai Elantra actually costing USD 70000 is a statistical impossibility. When this Hyundai Elantra's specs are fed to any of the project's trained supervised models, it produces an estimate that falls in the real-world and test dataset average price.

5.2 Reflection

The project explored a lot of what makes supervised learning versatile and accessible. It can be daunting for engineers who has not worked on any machine learning projects to tackle real-world problems with real-world data. In this project, supervised learning's accessibility is displayed with no less than two plug-and-play algorithms provided by Amazon SageMaker. These two algorithms' versatility of application proves that the barrier of entry to machine learning for engineers is not as difficult as the actual formal terms of machine learning evoke.

The project also showcases PyTorch, an open-source framework that does most of the heavy-lifting for machine learning engineers. With frameworks like this, experimentation on different architectures are made more accessible and faster. The project could have easily gone the other way around and set its objective on classifying the car model based on its price and other features and the tools used would still be sufficient, albeit with some parameter changes.

5.3 Improvement

The project found that the Used Cars Dataset did not have the actual year the car was manufactured or bought. This feature would have definitely impacted the *price* and undoubtedly be selected during feature selection. An improvement to the project could be made by augmenting the web crawler used to fill the Used Cars Dataset to include the manufacturing year of the cars.

Another approach is to modify the car *model* feature normalization to check for year inclusion in its value. For example, if *model* value is “**Cruze 2016**”, it could take the “**2016**” and set it as the manufacturing year feature of that entry in the dataset.

6 References

6.1 Brownlee, Jason. "Undersampling Algorithms for Imbalanced Classification." Machine Learning Mastery, 19 Jan. 2020, machinelearningmastery.com/undersampling-algorithms-for-imbalanced-classification/.

6.2 Bushaev, Vitaly. "Adam — Latest Trends in Deep Learning Optimization." Medium, Towards Data Science, 24 Oct. 2018, towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c.

6.3 Chakrabarty, Navoneel. "Application of Synthetic Minority Over-Sampling Technique (SMOTe) for Imbalanced Datasets." Medium, Towards AI, 16 July 2019, medium.com/towards-artificial-intelligence/application-of-synthetic-minority-over-sampling-technique-smote-for-imbalanced-data-sets-509ab55cfdaf.

6.4 Editor, Minitab Blog. "Regression Analysis: How Do I Interpret R-Squared and Assess the Goodness-of-Fit?" Minitab Blog, blog.minitab.com/blog/adventures-in-statistics-2/regression-analysis-how-do-i-interpret-r-squared-and-assess-the-goodness-of-fit.

6.5 Hudgeon, D., & Nichol, R. (2020). Machine learning for business: using Amazon SageMaker and Jupyter. Retrieved from https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost.html

6.6 Hudgeon, D., & Nichol, R. (2020). Machine learning for business: using Amazon SageMaker and Jupyter. Retrieved from https://docs.aws.amazon.com/sagemaker/latest/dg/linear-learner.html

6.7 Hudgeon, Doug, and Richard Nichol. "Machine Learning for Business: Using Amazon SageMaker and Jupyter." Amazon, Manning Publications Co., 2020, docs.aws.amazon.com/sagemaker/latest/dg/automatic-model-tuning.html.

6.8 “Mean Squared Error Loss Function: Peltarion Platform.” Peltarion.com, peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/mean-squared-error.

6.9 Reese, A. (2020, January 7). Used Cars Dataset. Retrieved from https://www.kaggle.com/austinreese/craigslist-carstrucks-data