OLD CAR PRICE PREDICTION

UNIVERSITY OF HYDERABAD

BY- **JAIVEER SINGH BANGAR**

EMAIL-jaiveerbangar@gmail.com

OLD CAR PRICE PREDICTION

An outlook:

The prices of new cars in the industry is fixed by the manufacturer with some added costs incurred by the Government in the form of taxes. So, customers buying a new car can be assured of the money they invest to be worthy. But due to the increased price of new cars and the incapability of customers to buy new cars due to the lack of funds, used cars sales are on a global increase.There is a need for a used car price prediction system to effectively decide the worthiness of the car using a variety of features. Even though there are websites that offers this service, their prediction method may not be the best. Besides, different models and systems may contribute on predicting power for a used car's actual market value. It is important to know their actual market value while both buying and selling.

Why is price prediction a burning problem ?

If we see price prediction is a burning problem these days because manual work has bound to be having errors and manually it is done by pricing experts who would go through the inspection report and decide the depreciation due to varied factors this was subject to human errors and a lot of manual intervention was needed.

How big is the market in India

Market size of used cars in India in 2019 was at 24.24 billion With an expected growth rate of 15.12% CAGR for the year 2022 to 2025, which is 1.3 times the new car market. Benefits like used cars cost at least 40% less than new and attract customers much more towards the used cars rather than the new cars. If we see data which is obtained from the national transport authority of India the number of cars registered between 2003 and 2013 has seen an increase of 234%. With tough economic conditions in India such as rising inflation and other factors it is highly likely that the sale of second hand cars in India is going to increase.

The India used car market was valued at USD 27 billion in 2020, and it is expected to reach USD 50 billion by 20206, registering a CAGR of 15% during the forecast period, 2021-2026.

The COVID-19 pandemic had a minimal impact on the industry. With the increased number of people preferring individual mobility and more finance options infused into the used car market, the market is set to grow considerably. Reduced cash inflow due to the pandemic has forced buyers to look for alternatives other than new cars, and the used car industry has great growth potential in these terms. With the sales and production of new vehicles hindered due to the pandemic, the immediate option for the buyers is the used car market.

The used car market evolved in the country, with the growth of the organized and semi-organized sales sectors. The pre-owned car market recorded sales of 4.4 million units in FY2020 as compared to only 2.8 million units of new passenger vehicles in the same year.

With the new BS-VI emission standards implemented by the Government of India, the technological cost of cars to meet the standards will be undertaken by consumers. Additionally, according to the MD and CEO of Mahindra First Choice Wheels (MFCW), the company's focus on reducing the production of diesel cars, with Maruti Suzuki's decision to exit the diesel car segment by April 2020, is also expected to increase the demand for compact diesel cars in the price and mileage sensitive Indian market.

Factors such as standardized dealership experience, high price experience, and high financing cost for used cars may hinder the growth of the used car market. Some of the major players dominating the market are OLX, Mahindra First Choice Wheels, Cars24, Maruti True Value, Hyundai H Promise, and Droom, among others.

Key Market Trends

The Organized Channel is Expected to Register a Higher CAGR

The organized segment in the India used car market is expected to witness a CAGR of 22.79% during the forecast period.

The organized sales channel witnessed significant growth in the last three years. This growth is driven by increased sales of used cars in metro cities and a rise in online sales platforms, such as CarDekho, Cars24, and Droom.

The majority of the OEMs have already entered the used car market, and those who have not entered the used car market in the nascent stages have also entered the market during the last five years. Renault started the pre-owned car business in 2015, and Nissan had introduced the firm in 2017.

Significant OEMs in India, such as Maruti Suzuki, Mahindra, Hyundai, and Toyota, and luxury car manufacturers, such as BMW, Audi, and JLR, have their used car networks.

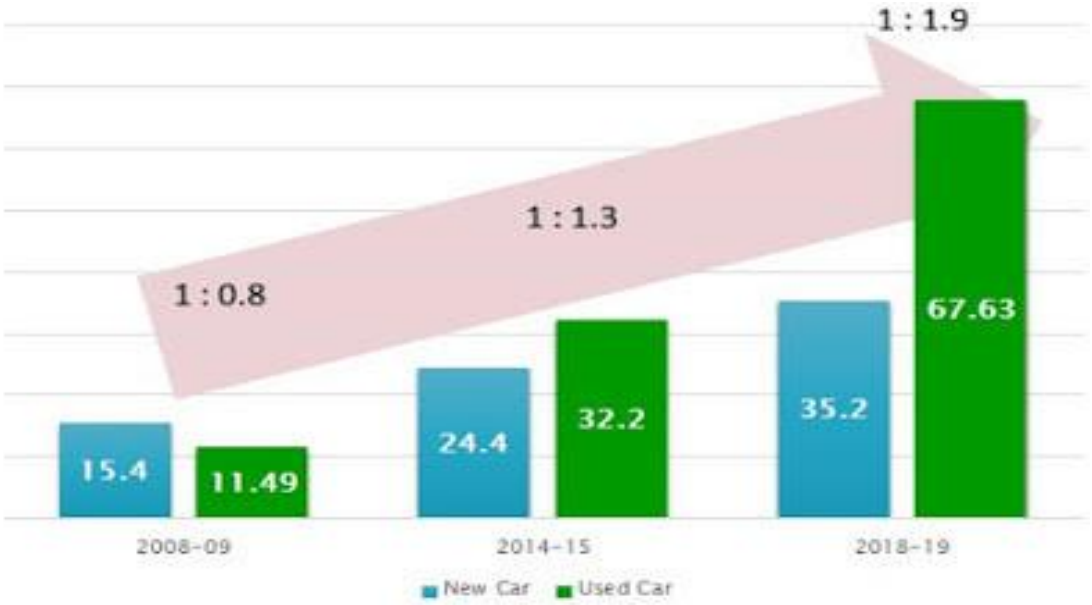
For instance, Mahindra First Choice Wheels recorded healthy annual sales of USD 51 million by the end of FY2020.

The company sold more than 250,000 used cars during the same period, and it aimed to increase this figure by 40% by 2020. Currently, MFCW has 1,100 outlets across India. By 2020, the company planned to increase the outlets to 1,200, out of which 11 outlets may be dedicated only to the sale of luxury cars.

The Indian car manufacturing giant Maruti Suzuki also entered the used cars market with its True Value chain, wherein the cars go through various quality tests before they are set up for sale.

Additionally, the online used car sellers registered good sales during 2017-2019. CarDekho has been a pioneer in the online used car market, recording a Y-o-Y growth rate of about 100% over the last three years.

Furthermore, consumers prefer safety, transparency, convenience, and negligible risk while purchasing used cars, which, in turn, indicates that the organized sector of used cars is projected to rapidly grow in terms of market share with immense growth potential.



Growing Demand for Luxury Used Cars

The used car market is witnessing a boom in the country, with the demand for luxury cars continually increasing.

Until few years, owning a luxury car used to be a dream for numerous consumers, owing to financial hurdles, but this is gradually changing as consumers can easily buy used luxury vehicles. Heavy depreciation in luxury car prices has made these vehicles a preferred choice in the used car market. As per OLX, used luxury vehicles priced over INR 15 lakh were the preferred choice among consumers. According to OLX, over 55,000 luxury cars (priced above INR 15 lakh) were listed on OLX every month, and the supply of premium cars jumped by over four times in 2017.

Some major factors driving the growth of the used luxury cars include the high rate of depreciation value of luxury cars, fast-growing young population, increasing disposable income of consumers (along with rapid urbanization), and growing internet penetration in non-metros.

As per automobile dealers, the demand for used luxury cars has been growing at approximately 35% - 40% on a year-on-year basis, as owners of luxury cars usually sell off their vehicles after a year or two years, as they desire for upgraded and better models. Additionally, apart from the reasons, majority buyers of these vehicles are from Tier-1 and Tier-2 cities.

On another note, over 75% of used cars purchased are sedans and hatchbacks due to their practicality and low-maintenance costs.

Competitive Landscape

The market for used cars in India is consolidated, with significant players holding the largest share of the market due to their business models and increased number of pre-owned car retail outlets. The major players include OLX, Mahindra First Choice Wheels, CARS24, Maruti True Value, and Hyundai H Promise.

Other players, such as Quikr, Honda Auto Terrace, Ford Assured, and Toyota U-Trust, have been expanding their outlets and operations in the local market. The unorganized sector occupies nearly 80% of the total market, which has been gradually shifting toward the organized sector in recent years.

A new company in the block, Big Boy Toyz, is in the market with an exclusive luxury, supercars, and hypercar line-up of preowned cars in its inventory. It is currently based in three Indian cities and deals with about 30 luxury carmakers.

Recent Developments

- In August 2021, Mahindra First Choice Wheels (MFCW) and CamCom, an AI-powered visual inspection solutions company, have partnered to offer the ability to inspect cars using Artificial Intelligence. With the partnership, Mahindra can inspect and make damage assessments for vehicles.
- In August 2021, Mercedes-Benz India has announced the introduction of a ‘direct customer to customer’ selling platform called ‘Marketplace’ to provide buyers and sellers of luxury pre-owned cars multiple benefits.
- In August 2021, Audi India announced its plan to expand Audi Approved Plus showrooms from 7 to 14 by end of 2021. which along with the 40 plus workshops countrywide will back up pre owned sales.
- In July 2021, Ola Expands Mobility Play With Used Car Marketplace ‘OlaCars’. OlaCars will directly compete with the likes of existing second hand retailing platforms such as CarDekho, CredR, OLX, Droom and Spiny which are currently the most funded companies in the space.

- In October 2020, Skoda Auto entered the India used car market with the "Certified Pre-Owned Program", offering a manufacturer-backed warranty to the vehicles certified by Skoda after 160 plus points of quality inspection and restoration wherever necessary. Toyota and Hyundai are also among the major OEMs in the used car market in the country.

If we see in many developed countries, it is quite common to lease a car rather than buying it outright, what is a lease?
It is a contract between the buyer and the seller, the buyer uses the car for a predefined set of days or kilometres whichever is decided between the two parties by paying a minimal fee, after the terms of the contract are completed the buyer then returns the car to the seller. This model of leasing has much more market than the new cars in developed countries.
We should also discuss the benefits of a used car over a new car.
To buy a new car a customer needs to shell out at least 30% more money than he would have been giving for the used car of the same model. Used cars also undergo very less depreciation compared to new cars.

Predicting the resale value of a car is not a simple task. Because the value of a used car depends on several factors such as the age of the car, the make of the car, the origin of the car, the mileage which it delivers et cetera. And moreover with rising fuel prices fuel economy is also particularly important. One more factor which has a significant impact on the price of a used car is its colour. Bright coloured cars tend to be cheaper unless they are imported or sports cars. In short we can say that the price of a used car depends upon multiple features and it's not an easy task to decide or predict the car price.
To be able to predict used cars market value can help both buyers and sellers.

Used car sellers (dealers): They are one of the biggest target groups that can be interested in the results of this study. If used car sellers better understand what makes a car desirable, what the key features are for a used car, then they may consider this knowledge and offer a better service.

Online pricing services: There are websites that offer an estimated value of a car. They may have a good prediction model. However, having a second model may help them to give a better prediction to their users. Therefore, the model developed in this study may help online web services that tell a used car's market value.

Individuals: There are lots of individuals who are interested in the used car market at some point in their life because they wanted to sell their car or buy a used car. In this process, it's a big corner to pay too much or sell less than its market value.

DATASET:

https://www.kaggle.com/saisaathvik/used-cars-dataset-from-cardekhocom

Data set is taken from kaggle and this data set has 21000 used cars scrapped from cardekho.com.

The data set holds 20000 items(cars) which have 13 features, those are- Name, selling price, new price, year, seller type, kilometers driven, owner type, fuel type, transmission type, mileage, engine capacity, max power, and number of seats.

Explanation of each feature:

1: Name of the car

2: Selling price- the price at which the car is available in the used car market.

3: New price- the price at which the car is available in the showroom.

4: Seller type- whether the car listed belongs to an owner or dealer, means some owner might have sold the car to a dealer and the dealer lists the car on his network, or the dealer will sell the car on behalf of the owner after taking a share of commission from the owner.

5: Kilometer driven- decide how much the car has been driven.

6: Owner type- whether the car is first owner, second owner etc

7: Fuel type- petrol, diesel or CNG.

8: Transmission- Automatic or Manual.

9: Mileage- fuel economy of the vehicle.

10: Engine capacity- its the cubic capacity of the engine(cc)

11: Max power- power is measured in brake horsepower(bhp)

12: Number of seats- 5 seater, 7 seater, 2 seater, 4 seater

13: Year- the manufacturing year of the car.

This data set will need a lot of feature engineering as there are a lot of features which will need updating etc. In the data set Selling price will be our dependent feature or we can say our target variable. The features which will have the highest impact in figuring out the price are, selling price, new price and kilometers driven. We can change fuel type to 1,2 using one hot encoding as it will be much easier for the algorithm to work with it. Features such as seller

type, owner type, transmission, name can be dropped. The biggest challenge in the dataset will be the null values of new price feature, which either can be refilled using a scraper from various other sites like carwale, zigwheels etc. to finetune the data more, i might use beautiful soup to extract data from other sites like carwale and car trade. This is just my first understanding of the dataset, which is not final and might change as I finetune the model with different parameters.
The tools which we will use for EDA will be Pandas and Matplotlib and seaborn.

The Models which we can use

A regression problem is when the output variable is a real or continuous value, such as “salary” or “weight”. Many different models can be used, the simplest is the linear regression. It tries to fit data with the best hyper-plane which goes through the points. Here in our data set the output variable is a real value that is the selling price of the car.

We will need to test out various algorithms of regression to reach a benchmark. At many instances linear regression is being used on such types of problems but the drawback of linear regression is its inability to handle multicollinearity which will be present in almost all the datasets of the real world. (we can measure collinearity using the Pearson correlation coefficient). Regression algorithms like XGboost have tend to given high accuracy results in such problem and we will explore it further in this project

Outlook of some regression techniques which we will try.

1. Linear Regression
Before we dive into the details of linear regression, you may be asking yourself why we are looking at this algorithm.
Isn’t it a technique from statistics?
Machine learning, more specifically the field of predictive modeling is primarily concerned with minimizing the error of a model or making the most accurate predictions possible, at the expense of explainability. In applied machine

learning we will borrow, reuse and steal algorithms from many different fields, including statistics and use them towards these ends.

As such, linear regression was developed in the field of statistics and is studied as a model for understanding the relationship between input and output numerical variables, but has been borrowed by machine learning. It is both a statistical algorithm and a machine learning algorithm.

Linear Regression is an attractive model because the representation is so simple.

The representation is a linear equation that combines a specific set of input values (x) the solution to which is the predicted output for that set of input values (y). As such, both the input values (x) and the output value are numeric.

The linear equation assigns one scale factor to each input value or column, called a coefficient and represented by the capital Greek letter Beta (B). One additional coefficient is also added, giving the line an additional degree of freedom (e.g. moving up and down on a two-dimensional plot) and is often called the intercept or the bias coefficient.

For example, in a simple regression problem (a single x and a single y), the form of the model would be:

y = B0 + B1*x

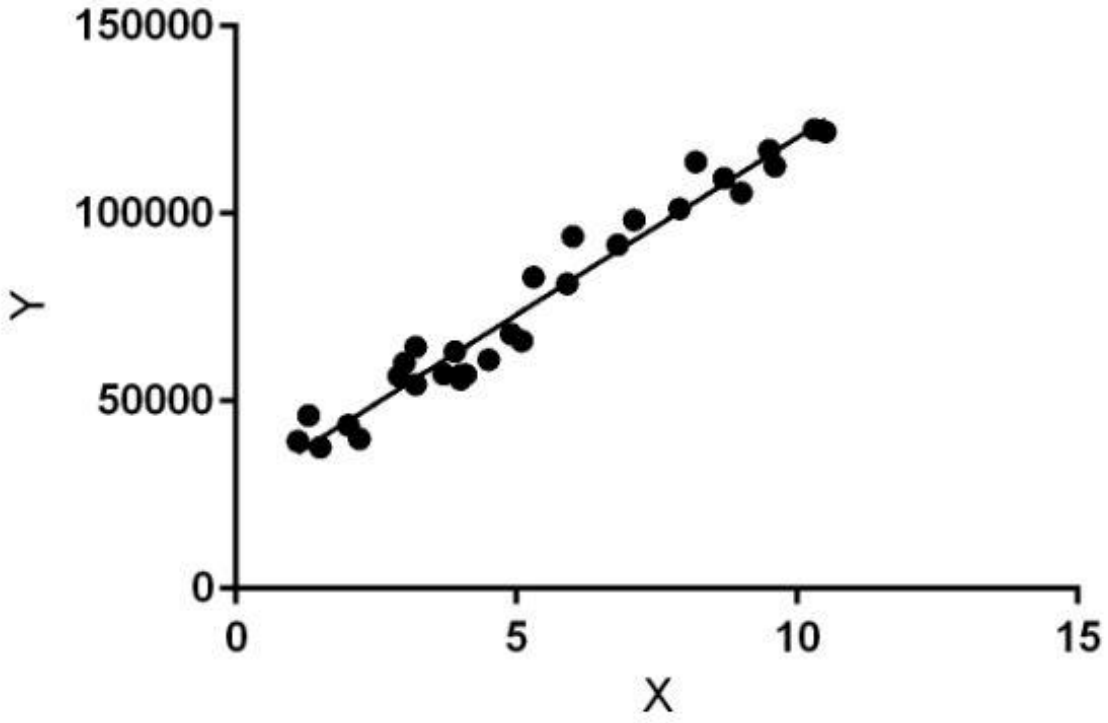
In higher dimensions when we have more than one input (x), the line is called a plane or a hyper-plane. The representation therefore is the form of the equation and the specific values used for the coefficients (e.g. B0 and B1 in the above example).

It is common to talk about the complexity of a regression model like linear regression. This refers to the number of coefficients used in the model.

When a coefficient becomes zero, it effectively removes the influence of the input

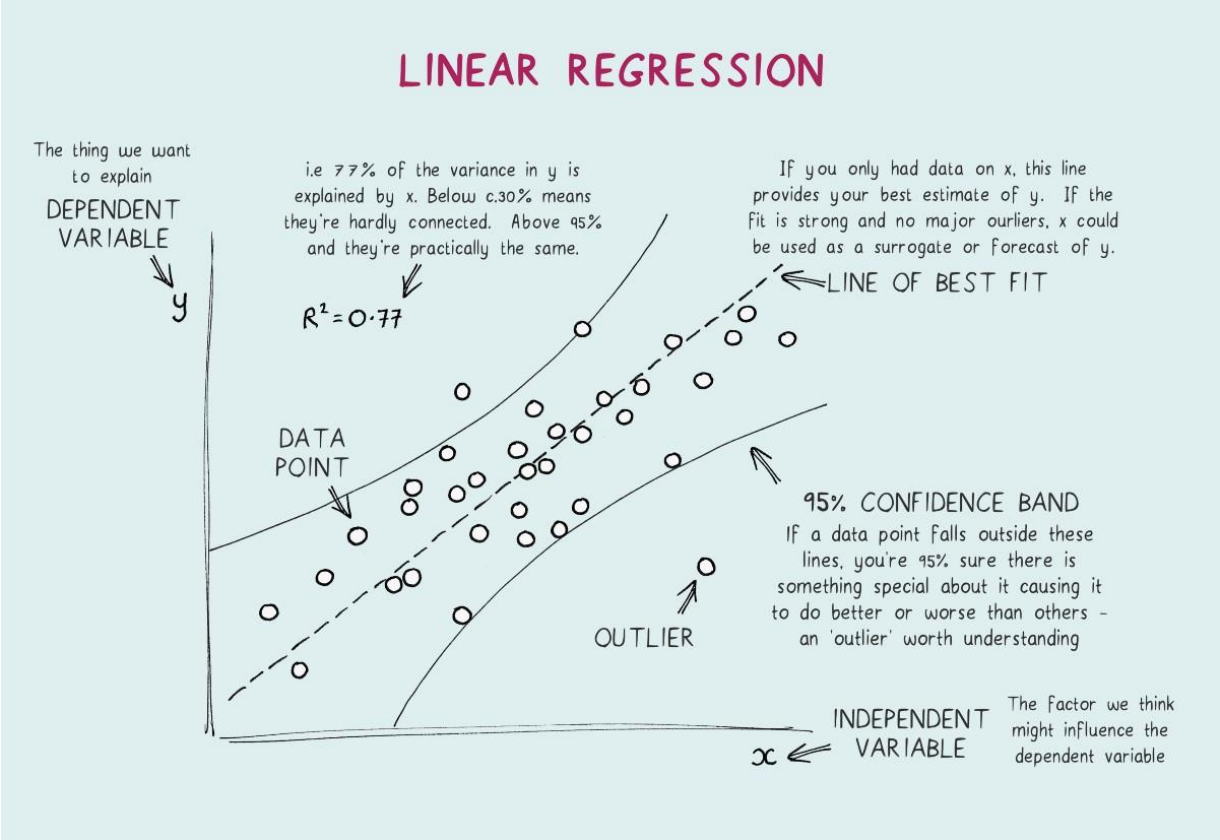
variable on the model and therefore from the prediction made from the model (0 * x = 0). This becomes relevant if you look at regularization methods that change the learning algorithm to reduce the complexity of regression models by putting pressure on the absolute size of the coefficients, driving some to zero.

Linear Regression is a machine learning algorithm based on supervised learning. It performs a regression task. Regression models a target prediction value based on independent variables. It is mostly used for finding out the relationship between variables and forecasting. Different regression models differ based on – the kind of relationship between dependent and independent variables, they are considering and the number of independent variables being used.



Linear regression performs the task to predict a dependent variable value (y) based on a given independent variable (x). So, this regression technique finds out a linear relationship between x (input) and y(output). Hence, the name is Linear Regression.

In the figure above, X (input) is the work experience and Y (output) is the salary of a person. The regression line is the best fit line for our model.



Advantages of Linear Regression

Simple implementation

Linear Regression is a very simple algorithm that can be implemented very easily to give satisfactory results.Furthermore, these models can be trained easily and efficiently even on systems with relatively low computational power when compared to other complex algorithms.Linear regression has a considerably lower time complexity when compared to some of the other machine learning algorithms.The mathematical equations of Linear regression are also fairly easy to understand and interpret.Hence Linear regression is very easy to master.

Performance on linearly seperable datasets

Linear regression fits linearly seperable datasets almost perfectly and is often used to find the nature of the relationship between variables.

Overfitting can be reduced by regularization

Overfitting is a situation that arises when a machine learning model fits a dataset very closely and hence captures the **noisy data** as well.This negatively impacts the performance of model and reduces its accuracy on the test set. Regularization is a technique that can be easily implemented and is capable of

effectively reducing the complexity of a function so as to reduce the risk of overfitting.

Disadvantages of Linear Regression

Prone to underfitting

Underfitting : A sitiuation that arises when a machine learning model fails to capture the data properly.This typically occurs when the hypothesis function cannot fit the data well.

Since linear regression assumes a linear relationship between the input and output varaibles, it fails to fit complex datasets properly. In most real life scenarios the relationship between the variables of the dataset isn't linear and hence a straight line doesn't fit the data properly. In such situations a more complex function can capture the data more effectively.Because of this most linear regression models have low accuracy.

Sensitive to outliers

Outliers of a data set are anomalies or extreme values that deviate from the other data points of the distribution.Data outliers can damage the performance of a machine learning model drastically and can often lead to models with low accuracy.

Outliers can have a very big impact on linear regression's performance and hence they must be dealt with appropriately before linear regression is applied on the dataset.

Linear Regression assumes that the data is independent

Very often the inputs aren't independent of each other and hence any **multicollinearity** must be removed before applying linear regression.

Conclusion

While the results produced by linear regression may seem impressive on linearly seperable datasets, it isn't recommended for most real world applications as it produces overly simplified results by assuming a linear relationship between the data.

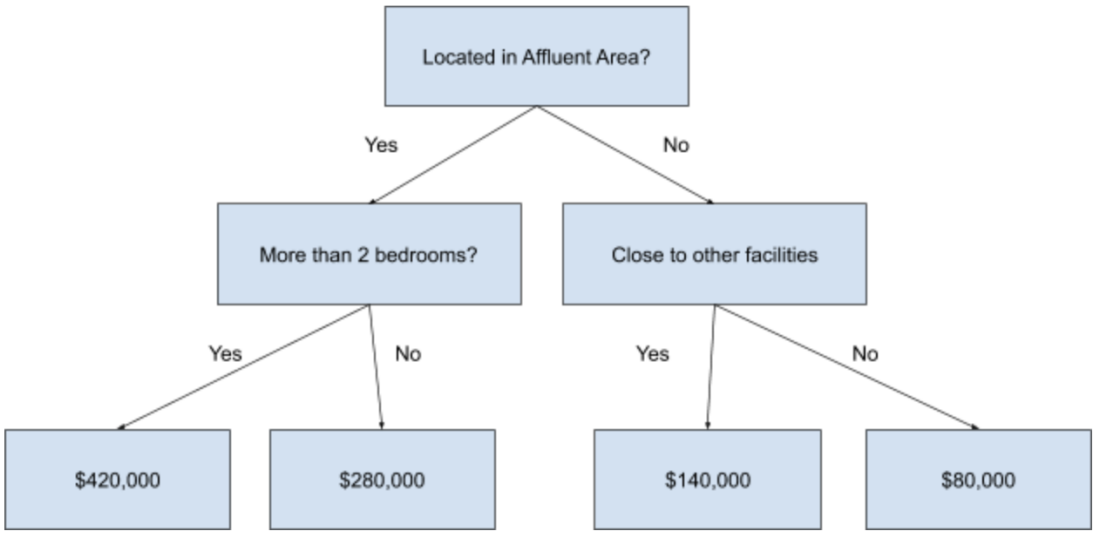
2. Random Forest

Random Forest is an ensemble learning based regression model. It uses a model called decision tree, specifically as the name suggests, multiple decision trees to generate the ensemble model which collectively produces a prediction. Random forest is one of the most popular algorithms for regression problems (i.e. predicting continuous outcomes) because of its simplicity and high accuracy. In this guide, we'll give you a gentle introduction to random forest and the reasons behind its high popularity.

How would random forest be described in layman's terms?

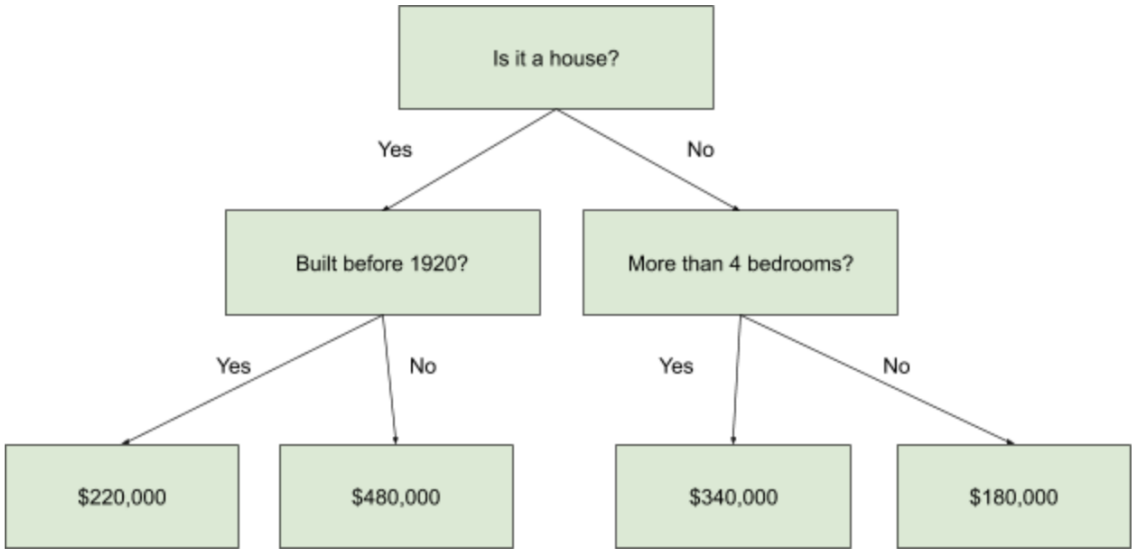
Let's start with an actual problem. Imagine you want to buy real estate, and you want to figure out what comprises a good deal so that you don't get taken advantage of.

The obvious thing to do would be to look at historic prices of houses sold in the area, then create some kind of decision criteria to summarize the average selling prices given the real-estate specification. You can use the decision chart to evaluate whether the listed price for the apartment you are considering is a bargain or not. It could look like this:



The chart represents a decision tree through a series of yes/no questions, which lead you from the real-estate description (“3 bedrooms”) to its historic average price. You can use the decision tree to predict what the expected price of a real estate would be, given its attributes.

However, you could come up with a distinctly different decision tree structure:



This would also be a valid decision chart, but with totally different decision criteria. These decisions are just as well-founded and show you information that was absent in the first decision tree.
The random forest regression algorithm takes advantage of the ‘wisdom of the crowds’. It takes multiple (but different) regression decision trees and makes them ‘vote’. Each tree needs to predict the expected price of the real estate based on the decision criteria it picked. Random forest regression then calculates the average of all of the predictions to generate a great estimate of what the expected price for a real estate should be.

What are the business use cases of random forest?

Random forest regression is used to solve a variety of business problems where the company needs to predict a continuous value:

1. Predict future prices/costs. Whenever your business is trading products or services (e.g. raw materials, stocks, labors, service offerings, etc.), you can use random forest regression to predict what the prices of these products and services will be in the future.
2. Predict future revenue. Use random forest regression to model your operations. For example, you can input your investment data (advertisement, sales materials, cost of hours worked on long-term enterprise deals, etc.) and your revenue data, and random forest will discover the connection between the input and output. This connection can be used to predict how much revenue you will generate based on the growth activity that you pick (marketing, direct to customer sales, enterprise sales, etc.) and how much you are willing to spend on it.
3. Compare performance. Imagine that you've just launched a new product line. The problem is, it's unclear whether the new product is attracting more (and higher spending) customers than the existing product line. Use random forest regression to determine how your new product compares to your existing ones.

Random forest regression is extremely useful in answering interesting and valuable business questions, but there are additional reasons why it is one of the most used machine learning algorithms.

What are the advantages of random forest for real production applications?

Random forest regression is a popular algorithm due to its many benefits in production settings:

4. Extremely high accuracy. Thanks to its 'wisdom of the crowds' approach, random forest regression achieves extremely high accuracies. It usually produces better results than other linear models, including linear regression and logistic regression.

5. Scales well. Computationally, the algorithm scales well when new features or samples are added to the dataset.
6. Interpretable. Although it is not as easily explainable as its underlying algorithm decision tree regression, random forests can be inspected to output the decision trees which were used in the final decision. The individual trees can be used to understand what the important decision nodes were, as well as prompt questions around what led to the final prediction.
7. Easy to use. Random forest works with both categorical and numerical input variables, so you spend less time one-hot encoding or labeling data. It's not sensitive to missing data, and it can handle outliers to a certain extent. Overall, it saves you time that would otherwise be spent cleaning data, which is usually the biggest step in any data science pipeline. This doesn't mean that you should skip the cleaning stage entirely: you will often obtain better performance by working the data into an appropriate shape. But random forest does make it easier to use and faster to deploy to reach the base model.

Machine learning approaches to random forest

Random forest is both a supervised learning algorithm *and* an ensemble algorithm.

It is supervised in the sense that during training, it learns the mappings between inputs and outputs. For example, an input feature (or independent variable) in the training dataset would specify that an apartment has "3 bedrooms" (feature: *number of bedrooms*) and this maps to the output feature (or target) that the apartment will be sold for "$200,000" (target: *price sold*).

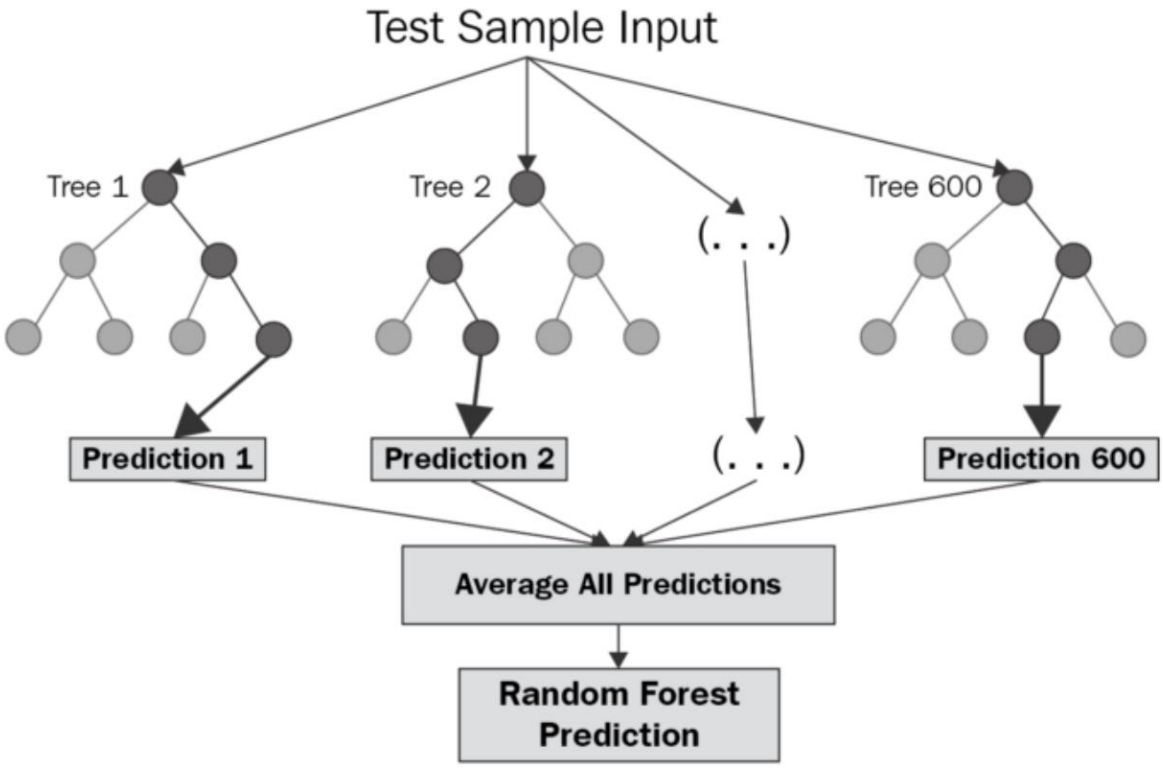
Ensemble algorithms combine multiple other machine learning algorithms, in order to make more accurate predictions than any underlying algorithm could on its own. In the case of random forest, it ensembles multiple decision trees into its final decision.

Random forest can be used on both regression tasks (predict continuous outputs, such as price) or classification tasks (predict categorical or discrete outputs). Here, we will take a deeper look at using random forest for regression predictions.

The random forest regression model

The random forest algorithm follows a two-step process:

8. Builds *n* decision tree regressors (estimators). The number of estimators *n* defaults to 100 in Scikit Learn (the machine learning Python library), where it is called *n_estimators*. The trees are built following the specified hyperparameters (e.g. minimum number of samples at the leaf nodes, maximum depth that a tree can grow, etc.).
9. Average prediction across estimators. Each decision tree regression predicts a number as an output for a given input. Random forest regression takes the average of those predictions as its 'final' output.



Pros of Random Forest

- Random Forests can be used for both classification and regression tasks.
- Random Forests work well with both categorical and numerical data. No scaling or transformation of variables is usually necessary.
- Random Forests implicitly perform feature selection and generate uncorrelated decision trees. It does this by choosing a random set of

features to build each decision tree. This also makes it a great model when you have to work with a high number of features in the data.

- Random Forests are not influenced by outliers to a fair degree. It does this by binning the variables.
- Random Forests can handle linear and non-linear relationships well.
- Random Forests generally provide high accuracy and balance the bias-variance trade-off well. Since the model's principle is to average the results across the multiple decision trees it builds, it averages the variance as well.

Cons of Random Forest

- Random Forests are not easily interpretable. They provide feature importance but it does not provide complete visibility into the coefficients as linear regression.
- Random Forests can be computationally intensive for large datasets.
- Random forest is like a black box algorithm, you have very little control over what the model does.

3. XGBoost

Extreme Gradient Boosting or XGBoost is one of the most popular machine learning models in current times. XGBoost is quite similar at the core to the original gradient boosting algorithm but features many additive features that significantly improve its performance such as built in support for regularization, parallel processing as well as giving additional hyperparameters to tune such as tree pruning, sub sampling and number of decision trees.

XGBoost is usually used to train gradient-boosted decision trees (GBDT) and other gradient boosted models. Random forests also use the same model representation and inference as gradient-boosted decision trees, but it is a different training algorithm. XGBoost can be used to train a standalone random forest. Also, random forest can be used as a base model for gradient boosting techniques.
Further, random forest is an improvement over bagging that helps in reducing the variance. Random forest builds trees in parallel, while in boosting, trees are built sequentially. Meaning, each of the trees is grown using information from previously grown trees, unlike bagging, where multiple copies of original training data are created and fit separate decision tree on each. This is the reason why XGBoost generally performs better than random forest.

When using gradient boosting for regression, where the weak learners are considered to be regression trees, each of the regression trees maps an input data point to one of its leaves that includes a continuous score. XGB minimises a regularised objective function that merges a convex loss function, which is based on the variation between the target outputs and the predicted outputs. The training then proceeds iteratively, adding new trees with the capability to predict the residuals as well as errors of prior trees that are then coupled with the previous trees to make the final prediction.

The data pre-processing steps for XGB include the following-

- Load the data
- Explore the data and remove the unneeded attributes
- Transform textual values to numeric
- Find and replace the missing values if needed
- Encoding the categorical data
- Break the dataset into training set as well as test set
- Perform feature scaling or data normalisation

Advantages:

- XGB consists of a number of hyper-parameters that can be tuned — a primary advantage over gradient boosting machines.
- XGBoost has an in-built capability to handle missing values.
- It provides various intuitive features, such as parallelisation, distributed computing, cache optimisation, and more.

Disadvantages:

- Like any other boosting method, XGB is sensitive to outliers.
- Unlike LightGBM, in XGB, one has to manually create dummy variable/ label encoding for categorical features before feeding them into the models.

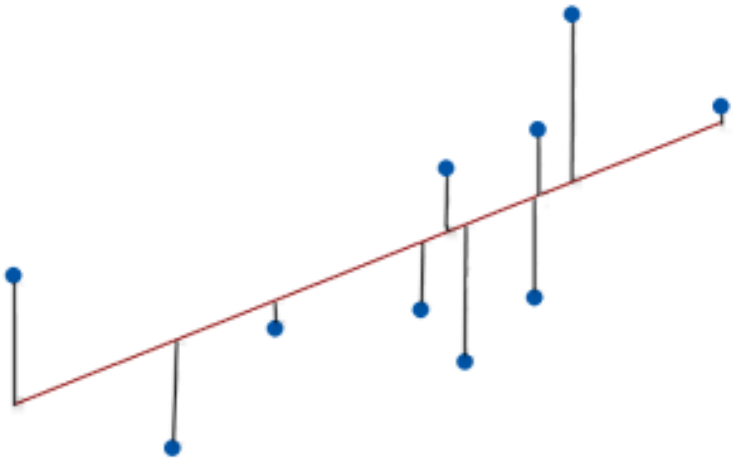
Metrics for Regression

In this section, we will take a closer look at the popular metrics for regression models and how to calculate them for your predictive modeling project.

Mean Squared Error

Mean squared error (MSE) measures the amount of error in statistical models. It assesses the average squared difference between the observed and predicted values. When a model has no error, the MSE equals zero. As model error increases, its value increases. The mean squared error is also known as the mean squared deviation (MSD).

For example, in regression, the mean squared error represents the average squared residual.



As the data points fall closer to the regression line, the model has less error, decreasing the MSE. A model with less error produces more precise predictions.

MSE Formula

The formula for MSE is the following.

$$MSE = \frac{\sum(y_i - \hat{y}_i)^2}{n}$$

Where:

- y_i is the i^{th} observed value.
- $\hat{y}_i$ is the corresponding predicted value.
- n = the number of observations.

The calculations for the mean squared error are similar to the variance. To find the MSE, take the observed value, subtract the predicted value, and square that difference. Repeat that for all observations. Then, sum all of those squared values and divide by the number of observations.
Notice that the numerator is the sum of the squared errors (SSE), which linear regression minimizes. MSE simply divides the SSE by the sample size.

Interpreting the Mean Squared Error

The MSE is the average squared distance between the observed and predicted values. Because it uses squared units rather than the natural data units, the interpretation is less intuitive.
Squaring the differences serves several purposes.
Squaring the differences eliminates negative values for the differences and ensures that the mean squared error is always greater than or equal to zero. It is almost always a positive value. Only a perfect model with no error produces an MSE of zero. And that doesn't occur in practice.
Additionally, squaring increases the impact of larger errors. These calculations disproportionately penalize larger errors more than smaller errors. This property is essential when you want your model to have smaller errors.
If you take the square root of the MSE, you obtain the root mean square error (RMSE), which does use the natural data units. In other words, MSE is analogous to the variance, whereas RMSE is akin to the standard deviation.

Root Mean Square Error

Root Mean Square Error (RMSE) is a standard way to measure the error of a model in predicting quantitative data. Formally it is defined as follows:

$$RMSE = \sqrt{\sum_{i=1}^{n} \frac{(\hat{y}_i - y_i)^2}{n}}$$

$$\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_n \text{ are predicted values}$$
$$y_1, y_2, \ldots, y_n \text{ are observed values}$$
$$n \text{ is the number of observations}$$

Let's try to explore why this measure of error makes sense from a mathematical perspective. Ignoring the division by n under the square root, the first thing we can notice is a resemblance to the formula for the Euclidean distance between two vectors in $\mathbb{R}^n$:

$$distance(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

This tells us heuristically that RMSE can be thought of as some kind of (normalized) distance between the vector of predicted values and the vector of observed values.

But why are we dividing by n under the square root here? If we keep n (the number of observations) fixed, all it does is rescale the Euclidean distance by a factor of $\sqrt{(1/n)}$. It's a bit tricky to see why this is the right thing to do, so let's delve in a bit deeper.

Imagine that our observed values are determined by adding random "errors" to each of the predicted values, as follows:

$$y_i = \hat{y}_i + \epsilon_i \text{ for } i = 1, \ldots, n$$
$$\epsilon_1, \ldots, \epsilon_n \text{ independent, identically distributed errors}$$

These errors, thought of as random variables, might have Gaussian distribution with mean μ and standard deviation σ, but any other distribution with a square-integrable PDF (*probability density function*) would also work. We want to think of $\hat{y}_i$ as an underlying physical quantity, such as the exact distance from Mars to the Sun at a particular point in time. Our observed quantity y_i would then be the distance from Mars to the Sun *as we measure it*, with some errors coming from mis-calibration of our telescopes and measurement noise from atmospheric interference.

The mean μ of the distribution of our errors would correspond to a persistent bias coming from mis-calibration, while the standard deviation σ would correspond to the amount of measurement noise. Imagine now that we know the mean μ of the distribution for our errors exactly and would like to estimate the standard deviation σ. We can see through a bit of calculation that:

$$\begin{aligned}
&\mathbb{E}\left[\frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{n}\right] \\
&= \mathbb{E}\left[\frac{\sum_{i=1}^{n}\epsilon_i^2}{n}\right] \\
&= \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}[\epsilon_i^2] \\
&= \mathbb{E}[\epsilon^2] \\
&= Var(\epsilon) + \mathbb{E}[\epsilon]^2 \\
&= \sigma^2 + \mu^2
\end{aligned}$$

Here E[…] is the expectation, and Var(…) is the variance. We can replace the average of the expectations $E[\varepsilon_i^2]$ on the third line with the $E[\varepsilon^2]$ on the fourth line where ε is a variable with the same distribution as each of the ε_i, because the errors ε_i are identically distributed, and thus their squares all have the same expectation.

Remember that we assumed we already knew μ exactly. That is, the persistent bias in our instruments is a known bias, rather than an unknown bias. So we might as well correct for this bias right off the bat by subtracting μ from all our raw observations. That is, we might as well suppose our errors are already distributed with mean μ = 0. Plugging this into the equation above and taking the square root of both sides then yields:

$$\sqrt{\mathbb{E}\left[\frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{n}\right]} = \sqrt{\sigma^2 + 0^2} = \sigma$$

Notice the left hand side looks familiar! If we removed the expectation E[…] from inside the square root, it is exactly our formula for RMSE form before. The central limit theorem tells us that as n gets larger, the variance of the quantity $\Sigma_i (\hat{y}_i - y_i)^2 / n = \Sigma_i (\varepsilon_i)^2 / n$ should converge to zero. In fact a sharper form of the central limit theorem tell us its variance should converge to 0 asymptotically like 1/n. This tells us that $\Sigma_i (\hat{y}_i - y_i)^2 / n$ is a good estimator for $E[\Sigma_i (\hat{y}_i - y_i)^2 / n] = \sigma^2$. But then RMSE is a good estimator for the standard deviation σ of the distribution of our errors!

We should also now have an explanation for the division by n under the square root in RMSE: it allows us to estimate the standard deviation σ of the error for a typical *single* observation rather than some kind of “total error”. By dividing by n, we keep this measure of error consistent as we move from a small collection of observations to a larger collection (it just becomes more accurate as we increase the number of observations). To phrase it another way, RMSE is a good way to answer the question: “How far off should we expect our model to be on its next prediction?”

To sum up our discussion, RMSE is a good measure to use if we want to estimate the standard deviation σ of a typical observed value from our model’s prediction, assuming that our observed data can be decomposed as:

Observed val=predicted val+predictably distribute random noise with mean zero

The random noise here could be anything that our model does not capture (e.g., unknown variables that might influence the observed values). If the noise is small, as estimated by RMSE, this generally means our model is good at predicting our observed data, and if RMSE is large, this generally means our model is failing to account for important features underlying our data.

RMSE in Data Science: Subtleties of Using RMSE

In data science, RMSE has a double purpose:

- To serve as a heuristic for training models
- To evaluate trained models for usefulness / accuracy

This raises an important question: What does it mean for RMSE to be “small”? We should note first and foremost that “small” will depend on our choice of units, and on the specific application we are hoping for. 100 inches is a big error in a building design, but 100 nanometers is not. On the other hand, 100 nanometers is a small error in fabricating an ice cube tray, but perhaps a big error in fabricating an integrated circuit.

For training models, it doesn’t really matter what units we are using, since all we care about during training is having a heuristic to help us decrease the error with each iteration. We care only about *relative* size of the error from one step to the next, not the absolute size of the error.

But in evaluating trained models in data science for usefulness / accuracy , we do care about units, because we aren’t just trying to see if we’re doing better than last time: we want to know if our model can actually help us solve a practical problem. The subtlety here is that evaluating whether RMSE is sufficiently small or not will depend on how accurate we need our model to be for our given application. There is never going to be a mathematical formula for this, because it depends on things like human intentions (“What are you intending to do with this model?”), risk aversion (“How much harm would be caused be if this model made a bad prediction?”), etc.

Besides units, there is another consideration too: “small” also needs to be measured relative to the type of model being used, the number of data points, and the history of training the model went through before you evaluated it for accuracy. At first this may sound counter-intuitive, but not when you remember the problem of over-fitting.

There is a risk of over-fitting whenever the number of parameters in your model is large relative to the number of data points you have. For example, if we are trying to predict one real quantity y as a function of another real quantity x, and our observations are (x_i, y_i) with $x_1 < x_2 < x_3 \ldots$, a general interpolation theorem tells us there is some polynomial f(x) of degree at most n+1 with $f(x_i) = y_i$ for i =

1, … , n. This means if we chose our model to be a degree n+1 polynomial, by tweaking the parameters of our model (the coefficients of the polynomial), we would be able to bring RMSE all the way down to 0. This is true regardless of what our y values are. In this case RMSE isn't really telling us anything about the accuracy of our underlying model: we were guaranteed to be able to tweak parameters to get RMSE = 0 as measured measured on our existing data points regardless of whether there is any relationship between the two real quantities at all.

But it's not only when the number of parameters exceeds the number of data points that we might run into problems. Even if we don't have an absurdly excessive amount of parameters, it may be that general mathematical principles together with mild background assumptions on our data guarantee us with a high probability that by tweaking the parameters in our model, we can bring the RMSE below a certain threshold. If we are in such a situation, then RMSE being below this threshold may not say anything meaningful about our model's predictive power.

Mean Absolute Error(MAE)

What is Absolute Error?

Absolute Error is the amount of error in your measurements. It is the difference between the measured value and "true" value. For example, if a scale states 90 pounds but you know your true weight is 89 pounds, then the scale has an absolute error of 90 lbs – 89 lbs = 1 lbs.

This can be caused by your scale not measuring the exact amount you are trying to measure. For example, your scale may be accurate to the nearest pound. If you weigh 89.6 lbs, the scale may "round up" and give you 90 lbs. In this case the absolute error is 90 lbs – 89.6 lbs = .4 lbs.

Formula

The formula for the absolute error (Δx) is:

$(\Delta x) = x_i - x,$

Where:

- x_i is the measurement,

- x is the true value.

Using the first weight example above, the absolute error formula gives the same result:

(Δx) = 90 lbs – 89 lbs = 1 lb.

Sometimes you'll see the formula written with the absolute value symbol (these bars: | |). This is often used when you're dealing with multiple measurements:

$(\Delta x) = |x_i - x|,$

The absolute value symbol is needed because sometimes the measurement will be smaller, giving a negative number. For example, if the scale measured 89 lbs and the true value was 95 lbs then you would have a difference of 89 lbs – 95 lbs = -6 lbs. On its own, a negative value is fine (-6 just means "six units below") but the problem comes when you're trying to add several values, some of which are positive and some are negative. For example, let's say you have:

- 89 lbs – 95 lbs = -6 lbs and
- 98 lbs – 92 lbs = 6 lbs

On their own, both measurements have absolute errors of 6 lbs. If you add them together, you should get a total of 12 lbs of error, but because of that negative sign you'll actually get -6 lbs + 6 lbs = 0 lbs, which makes no sense at all — after all, there was a pretty big error (12 lbs) which has somehow become 0 lbs of error. We can solve this by taking the absolute value of the results and then adding:

|-6 lbs| + |6 lbs| = 12 lbs.

The Mean Absolute Error(MAE) is the average of all absolute errors. The formula is:

$$\mathbf{MAE} = \frac{1}{n}\sum_{i=1}^{n} |\boldsymbol{x}_i - \boldsymbol{x}|$$

Where:

- n = the number of errors,
- Σ = summation symbol (which means "add them all up"),

- $|x_i - x|$ = the absolute errors.

Challenges

The car price prediction poses several challenges

1: Item variety and complexity- The number of different car models produced by manufacturers is quite important and continuously grows larger as new cars are launched every year. Car models evolve, and the same car is available in variants belonging to different power classes, not to mention the vast range of practical options and accessories. There are a high number of factors that contribute to item pricing and it is thus hard to choose a modeling approach.

2: The environment evolves rapidly- Building a predictor that can stand the test of time and not be useful only in an extremely limited time window is a considerable challenge.

3: Imbalanced datasets- Any car dataset that reflects the distribution of real-world markets will be imbalanced in terms of manufacturers and car models. The scarcity of sale announcements for many types of cars makes it difficult to define a single machine learning model that performs reasonably well for a wide range of car models. Finding ways to use similarities between different cars is another necessity of the predictor.

4: Unreliable data- Digital marketplaces show us the seller's proposed price. This estimation will in most cases adhere to real market trends and be a reasonable price tag, but we won't know the final selling price of the item after off-platform negotiations. It is also difficult to detect offers that present large inaccuracies in the estimation process. For these reasons, we should be aware that these crowd-based datasets are inherently noisy. The ML model must hence account for high output variability and outliers.

EDA(Exploratory Data Analysis)

There was a choice of two datasets in Kaggle, one was a raw data set and the second one was cleaned, for the purpose of data cleaning and EDA, I choose the raw data set as it will provide me with much more features and variables to play with.

first step was to drop all the unnecessary values in the dataset

Source.Name,web-scraper-order,web-scraper-start-url,mileage,new-price,max_power,seats.

new price basically consisted of over 50 percent null values and moreover our target variable will be the selling price, (for some layperson explanation- if we need to check for new car price, we don't need any model for it, we can simply go to websites like car Dekho etc. and simply putting the details we will be able to get the new price. Here the model aim is to deciding the used car price.)

other features like mileage and max power are irrelevant. why ?? because mileage of vehicle is highly affected by the fuel type and engine capacity, a diesel vehicle gives a higher mileage than a petrol or CNG vehicle, so keeping the fuel type feature will serve the purpose for the model.

I have created categorical values to numerical using pandas dummy command (say, one categorical variable has n values, while dummy encoding converts it into n-1 variables, whereas one hot encoding created it into n variables. if we have k categorical variables, each of which has n values. one hot encoding ends up with KN variables, while dummy encoding ends up with KN-K variables.)

For 'name' we dropped all the rows which held the null values and the rest we simply sorted using the mean function in individual column. I also created an age column by subtracting the manufacturing year from 2022 and thendropped the manufacturing year.

The correlation matrix and graphs gave us the following observations

1: selling price which is our target variable is positively correlated with engine capacity, automatic transmission, and diesel vehicle means that these features affect the selling price of vehicle positively, an automatic vehicle fetches better

price than a manual, similarly a higher capacity engine is worth more in the market, and diesel vehicle will fetch a better price than its petrol counterpart.

2: We have a strong negative correlation between selling price and kilometers driven by a car, age of the vehicle, and transmission as selling price of a manual vehicle is pretty less,similarly a higher mileage driven car is fetching less price, and same is the case with age, higher the age of vehicle, lesser is its price. we should try to see these relations with selling price more graphically to get more clarity.

3: Inverse relation of age with selling price, as age increases, selling price decreases

4: Engine capacity directly affects the price here, but after 3000cc it does not have much impact, as above 3000 cc it's the sports car territory, where the customers are limited and are not bothered about the engine capacity of their sports car.

In [ ]:

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

In [ ]:

```
import pandas as pd
```

In [ ]:

```
path="/content/drive/MyDrive/Cardekho_Extract.csv"
df=pd.read_csv(path)
df.describe
```

Out[ ]:

```
<bound method NDFrame.describe of                                Source.Name  ...    seats
0           cardekho_extract(0-2000).csv  ...   Seats5
1           cardekho_extract(0-2000).csv  ...   Seats5
2           cardekho_extract(0-2000).csv  ...   Seats5
3           cardekho_extract(0-2000).csv  ...   Seats5
4           cardekho_extract(0-2000).csv  ...   Seats5
...                                  ...  ...      ...
20021  cardekho_extract(5001-10000).csv  ...   Seats5
20022  cardekho_extract(5001-10000).csv  ...   Seats7
20023  cardekho_extract(5001-10000).csv  ...   Seats5
20024  cardekho_extract(5001-10000).csv  ...   Seats7
20025  cardekho_extract(5001-10000).csv  ...   Seats5

[20026 rows x 16 columns]>
```

In [ ]:

```
df.head(5)
```

Out[ ]:

	Source.Name	web-scraper-order	web-scraper-start-url	full_name	selling_price	new-price	year	seller_type
0	cardekho_extract(0-2000).csv	1611917819-1662	https://www.cardekho.com/used-car-details/used...	Maruti Alto Std	1.2 Lakh*	NaN	2012.0	Individual
1	cardekho_extract(0-2000).csv	1611918361-1902	https://www.cardekho.com/used-car-details/used...	Hyundai Grand i10 Asta	5.5 Lakh*	New Car (On-Road Price) : Rs.7.11-7.48 Lakh*	2016.0	Individual
2	cardekho_extract(0-2000).csv	1611917012-1306	https://www.cardekho.com/used-car-details/used...	Hyundai i20 Asta	2.15 Lakh*	NaN	2010.0	Individual
3	cardekho_extract(0-2000).csv	1611917695-1607	https://www.cardekho.com/used-car-details/used...	Maruti Alto K10 2010-2014 VXI	2.26 Lakh*	NaN	2012.0	Individual
4	cardekho_extract(0-2000).csv	1611914861-367	https://www.cardekho.com/used-car-details/used...	Ford Ecosport 2015-2021 1.5 TDCi Titanium BSIV	5.7 Lakh*	New Car (On-Road Price) : Rs.10.14-13.79 Lakh*	2015.0	Dealer

In [ ]:

```
## we dont need these and should be dropped
df.drop(['Source.Name','web-scraper-order','web-scraper-start-url','mileage','new-price',
'max_power','seats'], axis=1, inplace=True)
```

In [ ]:

```
#Top 5 rows
df.head(5)
```

Out[ ]:

	full_name	selling_price	year	seller_type	km_driven	owner_type	fuel_type	transmission_type	engine
0	Maruti Alto Std	1.2 Lakh*	2012.0	Individual	1,20,000 kms	First Owner	Petrol	Manual	Engine796 CC
1	Hyundai Grand i10 Asta	5.5 Lakh*	2016.0	Individual	20,000 kms	First Owner	Petrol	Manual	Engine1197 CC
2	Hyundai i20 Asta	2.15 Lakh*	2010.0	Individual	60,000 kms	First Owner	Petrol	Manual	Engine1197 CC
3	Maruti Alto K10 2010-2014 VXI	2.26 Lakh*	2012.0	Individual	37,000 kms	First Owner	Petrol	Manual	Engine998 CC
4	Ford Ecosport 2015-2021 1.5 TDCi Titanium BSIV	5.7 Lakh*	2015.0	Dealer	30,000 kms	First Owner	Diesel	Manual	Engine1498 CC

In [ ]:

```
#lets check how many null values we have
df.isnull().sum()
```

Out[ ]:

```
full_name            46
selling_price        46
year                 46
seller_type          46
km_driven            46
owner_type           46
fuel_type            46
transmission_type    46
engine              105
dtype: int64
```

In [ ]:

```
## either we can drop full_name also but i would like to change it to name of manufacture
r column
fulname = df["full_name"].str.upper()
df['full_name'] = fulname
new = df["full_name"].str.split(" ",n=1,expand=True)

## making separate name column
df["Name"]= new[0]

df.drop('full_name', inplace= True, axis=1)
df.head(5)
```

Out[ ]:

	selling_price	year	seller_type	km_driven	owner_type	fuel_type	transmission_type	engine	Name
0	1.2 Lakh*	2012.0	Individual	1,20,000 kms	First Owner	Petrol	Manual	Engine796 CC	MARUTI
1	5.5 Lakh*	2016.0	Individual	20,000 kms	First Owner	Petrol	Manual	Engine1197 CC	HYUNDAI
2	2.15 Lakh*	2010.0	Individual	60,000 kms	First Owner	Petrol	Manual	Engine1197 CC	HYUNDAI

	selling_price	year	seller_type	km_driven	owner_type	fuel_type	transmission_type	engine	Name
3	2.26 Lakh*	2012.0	Individual	37,000 kms	First Owner	Petrol	Manual	Engine998 CC	MARUTI
4	5.7 Lakh*	2015.0	Dealer	30,000 kms	First Owner	Diesel	Manual	Engine1498 CC	FORD

Next lets remove the alphabets from alphanumeric strings

In [ ]:

```
new = df["selling_price"].str.split(" ", n = 1, expand = True)
df["selling_price"]= new[0]

new = df["km_driven"].str.split(" ", n = 1, expand = True)
df["km_driven"]= new[0]

new = df["engine"].str.split("ne", n = 1, expand = True)
df["engine"]= new[1]
new = df["engine"].str.split(" ", n = 1, expand = True)
df["engine"]= new[0]

df.head()
```

Out[ ]:

	selling_price	year	seller_type	km_driven	owner_type	fuel_type	transmission_type	engine	Name
0	1.2	2012.0	Individual	1,20,000	First Owner	Petrol	Manual	796	MARUTI
1	5.5	2016.0	Individual	20,000	First Owner	Petrol	Manual	1197	HYUNDAI
2	2.15	2010.0	Individual	60,000	First Owner	Petrol	Manual	1197	HYUNDAI
3	2.26	2012.0	Individual	37,000	First Owner	Petrol	Manual	998	MARUTI
4	5.7	2015.0	Dealer	30,000	First Owner	Diesel	Manual	1498	FORD

Converting categorical value to numerical values and creating a new df so we can compare

In [ ]:

```
## converting transmission to boolean
transmission= pd.get_dummies(df['transmission_type'])
newdf= pd.concat([df,transmission],axis=1)
newdf.drop('transmission_type',inplace=True,axis=1)
newdf.head(5)
```

Out[ ]:

	selling_price	year	seller_type	km_driven	owner_type	fuel_type	engine	Name	Automatic	Manual
0	1.2	2012.0	Individual	1,20,000	First Owner	Petrol	796	MARUTI	0	1
1	5.5	2016.0	Individual	20,000	First Owner	Petrol	1197	HYUNDAI	0	1
2	2.15	2010.0	Individual	60,000	First Owner	Petrol	1197	HYUNDAI	0	1
3	2.26	2012.0	Individual	37,000	First Owner	Petrol	998	MARUTI	0	1
4	5.7	2015.0	Dealer	30,000	First Owner	Diesel	1498	FORD	0	1

In [ ]:

```
## convert fuel to boolean
fuel = pd.get_dummies(newdf['fuel_type'])
newdf = pd.concat([newdf,fuel],axis=1)
newdf.drop('fuel_type', inplace=True, axis=1)
newdf.head(5)
```

Out[ ]:

	selling_price	year	seller_type	km_driven	owner_type	engine	Name	Automatic	Manual	CNG	Diesel	Electric	LP
0	1.2	2012.0	Individual	1,20,000	First Owner	796	MARUTI	0	1	0	0	0	
1	5.5	2016.0	Individual	20,000	First Owner	1197	HYUNDAI	0	1	0	0	0	
2	2.15	2010.0	Individual	60,000	First Owner	1197	HYUNDAI	0	1	0	0	0	
3	2.26	2012.0	Individual	37,000	First Owner	998	MARUTI	0	1	0	0	0	
4	5.7	2015.0	Dealer	30,000	First Owner	1498	FORD	0	1	0	1	0	

In [ ]:

```
seller = pd.get_dummies(newdf['seller_type'])
newdf = pd.concat([newdf,seller],axis=1)
newdf.drop('seller_type', inplace=True, axis=1)

owner = pd.get_dummies(newdf['owner_type'])
newdf = pd.concat([newdf,owner],axis=1)
newdf.drop('owner_type', inplace=True, axis=1)

newdf.head(5)
```

Out[ ]:

	selling_price	year	km_driven	engine	Name	Automatic	Manual	CNG	Diesel	Electric	LPG	Petrol	Dealer	Individ
0	1.2	2012.0	1,20,000	796	MARUTI	0	1	0	0	0	0	1	0	
1	5.5	2016.0	20,000	1197	HYUNDAI	0	1	0	0	0	0	1	0	
2	2.15	2010.0	60,000	1197	HYUNDAI	0	1	0	0	0	0	1	0	
3	2.26	2012.0	37,000	998	MARUTI	0	1	0	0	0	0	1	0	
4	5.7	2015.0	30,000	1498	FORD	0	1	0	1	0	0	0	1	

In [ ]:

```
newdf.dtypes
```

Out[ ]:

```
selling_price        object
year                float64
km_driven            object
engine               object
Name                 object
Automatic             uint8
Manual                uint8
CNG                   uint8
Diesel                uint8
Electric              uint8
LPG                   uint8
Petrol                uint8
Dealer                uint8
Individual            uint8
Trustmark Dealer      uint8
First Owner           uint8
Second Owner          uint8
Third Owner           uint8
dtype: object
```

In [ ]:

```
newdf['selling_price'] = newdf['selling_price'].astype(str)
SP = []
for m in newdf['selling_price']:
    if "*" in m :
        m =float(m.replace(',', '').replace('*', ''))/100000
    SP.append(float(m))
```

```
newdf['selling_price'] = SP
newdf.head(5)
```

Out[ ]:

	selling_price	year	km_driven	engine	Name	Automatic	Manual	CNG	Diesel	Electric	LPG	Petrol	Dealer	Individ
0	1.20	2012.0	1,20,000	796	MARUTI	0	1	0	0	0	0	1	0	
1	5.50	2016.0	20,000	1197	HYUNDAI	0	1	0	0	0	0	1	0	
2	2.15	2010.0	60,000	1197	HYUNDAI	0	1	0	0	0	0	1	0	
3	2.26	2012.0	37,000	998	MARUTI	0	1	0	0	0	0	1	0	
4	5.70	2015.0	30,000	1498	FORD	0	1	0	1	0	0	0	1	

In [ ]:

```
newdf.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20026 entries, 0 to 20025
Data columns (total 18 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   selling_price     19980 non-null  float64
 1   year              19980 non-null  float64
 2   km_driven         19980 non-null  object 
 3   engine            19877 non-null  object 
 4   Name              19980 non-null  object 
 5   Automatic         20026 non-null  uint8  
 6   Manual            20026 non-null  uint8  
 7   CNG               20026 non-null  uint8  
 8   Diesel            20026 non-null  uint8  
 9   Electric          20026 non-null  uint8  
 10  LPG               20026 non-null  uint8  
 11  Petrol            20026 non-null  uint8  
 12  Dealer            20026 non-null  uint8  
 13  Individual        20026 non-null  uint8  
 14  Trustmark Dealer  20026 non-null  uint8  
 15  First Owner       20026 non-null  uint8  
 16  Second Owner      20026 non-null  uint8  
 17  Third Owner       20026 non-null  uint8  
dtypes: float64(2), object(3), uint8(13)
memory usage: 1.0+ MB
```

In [ ]:

```
newdf['km_driven'] = newdf['km_driven'].astype(str)
KM = []
for m in newdf['km_driven']:
    KM.append(m.replace(',', ''))
newdf['km_driven'] = KM
newdf.head(5)
```

Out[ ]:

	selling_price	year	km_driven	engine	Name	Automatic	Manual	CNG	Diesel	Electric	LPG	Petrol	Dealer	Individ
0	1.20	2012.0	120000	796	MARUTI	0	1	0	0	0	0	1	0	
1	5.50	2016.0	20000	1197	HYUNDAI	0	1	0	0	0	0	1	0	
2	2.15	2010.0	60000	1197	HYUNDAI	0	1	0	0	0	0	1	0	
3	2.26	2012.0	37000	998	MARUTI	0	1	0	0	0	0	1	0	
4	5.70	2015.0	30000	1498	FORD	0	1	0	1	0	0	0	1	

In [ ]:

```
newdf.isnull().sum()
```

Out[ ]:

```
selling_price        46
year                 46
km_driven             0
engine              149
Name                 46
Automatic             0
Manual                0
CNG                   0
Diesel                0
Electric              0
LPG                   0
Petrol                0
Dealer                0
Individual            0
Trustmark Dealer      0
First Owner           0
Second Owner          0
Third Owner           0
dtype: int64
```

In [ ]:

```
## we replace the null values of selling price column with its mean
newdf['selling_price'].fillna((newdf['selling_price'].mean()),inplace=True)
```

In [ ]:

```
## we replace the null values of year column with its mean
newdf['year'].fillna((newdf['year'].mean()),inplace=True)
```

In [ ]:

```
## converting engine column to float from object and replacing its null values with its m
ean
newdf['engine'] = pd.to_numeric(newdf['engine'], downcast="float")
newdf['engine'].fillna((newdf['engine'].mean()),inplace=True)
```

In [ ]:

```
newdf.isnull().sum()
```

Out[ ]:

```
selling_price        0
year                 0
km_driven            0
engine               0
Name                46
Automatic            0
Manual               0
CNG                  0
Diesel               0
Electric             0
LPG                  0
Petrol               0
Dealer               0
Individual           0
Trustmark Dealer     0
First Owner          0
Second Owner         0
Third Owner          0
dtype: int64
```

In [ ]:

```
## we should not assume any value to name, so we can drop the rows in which name column h
```

```
as null values
newdf.dropna(inplace=True)
```

In [ ]:

```
#As we can see, we have now 0 null values and our data is almost ready
newdf.isnull().sum()
```

Out[ ]:

```
selling_price       0
year                0
km_driven           0
engine              0
Name                0
Automatic           0
Manual              0
CNG                 0
Diesel              0
Electric            0
LPG                 0
Petrol              0
Dealer              0
Individual          0
Trustmark Dealer    0
First Owner         0
Second Owner        0
Third Owner         0
dtype: int64
```

In [ ]:

```
df.describe
```

Out[ ]:

```
<bound method NDFrame.describe of       selling_price    year seller_type  ... transmissi
on_type engine      Name
0               1.2  2012.0  Individual  ...            Manual    796    MARUTI
1               5.5  2016.0  Individual  ...            Manual   1197   HYUNDAI
2              2.15  2010.0  Individual  ...            Manual   1197   HYUNDAI
3              2.26  2012.0  Individual  ...            Manual    998    MARUTI
4               5.7  2015.0      Dealer  ...            Manual   1498      FORD
...             ...     ...         ...  ...               ...    ...       ...
20021           6.5  2017.0      Dealer  ...            Manual   1364    TOYOTA
20022          9.25  2019.0      Dealer  ...            Manual   1373    MARUTI
20023          4.25  2015.0      Dealer  ...            Manual   1498     SKODA
20024         12.25  2016.0      Dealer  ...            Manual   2179  MAHINDRA
20025            12  2019.0      Dealer  ...         Automatic   1497     HONDA

[20026 rows x 9 columns]>
```

In [ ]:

```
newdf.describe
```

Out[ ]:

```
<bound method NDFrame.describe of        selling_price    year km_driven  ...  First Owne
r Second Owner  Third Owner
0               1.20  2012.0    120000  ...            1            0            0
1               5.50  2016.0     20000  ...            1            0            0
2               2.15  2010.0     60000  ...            1            0            0
3               2.26  2012.0     37000  ...            1            0            0
4               5.70  2015.0     30000  ...            1            0            0
...              ...     ...       ...  ...          ...          ...          ...
20021           6.50  2017.0     69480  ...            1            0            0
20022           9.25  2019.0     18000  ...            1            0            0
20023           4.25  2015.0     67000  ...            1            0            0
20024          12.25  2016.0   3800000  ...            1            0            0
20025          12.00  2019.0     13000  ...            1            0            0

[19980 rows x 18 columns]>
```

In [ ]:

```
newdf.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 19980 entries, 0 to 20025
Data columns (total 18 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   selling_price     19980 non-null  float64
 1   year              19980 non-null  float64
 2   km_driven         19980 non-null  object 
 3   engine            19980 non-null  float32
 4   Name              19980 non-null  object 
 5   Automatic         19980 non-null  uint8  
 6   Manual            19980 non-null  uint8  
 7   CNG               19980 non-null  uint8  
 8   Diesel            19980 non-null  uint8  
 9   Electric          19980 non-null  uint8  
 10  LPG               19980 non-null  uint8  
 11  Petrol            19980 non-null  uint8  
 12  Dealer            19980 non-null  uint8  
 13  Individual        19980 non-null  uint8  
 14  Trustmark Dealer  19980 non-null  uint8  
 15  First Owner       19980 non-null  uint8  
 16  Second Owner      19980 non-null  uint8  
 17  Third Owner       19980 non-null  uint8  
dtypes: float32(1), float64(2), object(2), uint8(13)
memory usage: 1.1+ MB
```

In [ ]:

```
# it seems we still had the km driven column as object, so we must change it to float
newdf['km_driven']=newdf.km_driven.astype(float)
```

In [ ]:

```
newdf.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 19980 entries, 0 to 20025
Data columns (total 18 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   selling_price     19980 non-null  float64
 1   year              19980 non-null  float64
 2   km_driven         19980 non-null  float64
 3   engine            19980 non-null  float32
 4   Name              19980 non-null  object 
 5   Automatic         19980 non-null  uint8  
 6   Manual            19980 non-null  uint8  
 7   CNG               19980 non-null  uint8  
 8   Diesel            19980 non-null  uint8  
 9   Electric          19980 non-null  uint8  
 10  LPG               19980 non-null  uint8  
 11  Petrol            19980 non-null  uint8  
 12  Dealer            19980 non-null  uint8  
 13  Individual        19980 non-null  uint8  
 14  Trustmark Dealer  19980 non-null  uint8  
 15  First Owner       19980 non-null  uint8  
 16  Second Owner      19980 non-null  uint8  
 17  Third Owner       19980 non-null  uint8  
dtypes: float32(1), float64(3), object(1), uint8(13)
memory usage: 1.1+ MB
```

In [ ]:

```
## just adding an age column which will denote the age of vehicle
newdf['current year']=2022
```

In [ ]:

```
newdf['age']=newdf['current year']-newdf['year']
```

In [ ]:

```
newdf.drop(['year','current year'], axis=1, inplace=True)
```

In [ ]:

```
newdf.head(5)
```

Out[ ]:

	selling_price	km_driven	engine	Name	Automatic	Manual	CNG	Diesel	Electric	LPG	Petrol	Dealer	Individual	Trus
0	1.20	120000.0	796.0	MARUTI	0	1	0	0	0	0	1	0	1	
1	5.50	20000.0	1197.0	HYUNDAI	0	1	0	0	0	0	1	0	1	
2	2.15	60000.0	1197.0	HYUNDAI	0	1	0	0	0	0	1	0	1	
3	2.26	37000.0	998.0	MARUTI	0	1	0	0	0	0	1	0	1	
4	5.70	30000.0	1498.0	FORD	0	1	0	1	0	0	0	1	0	

In [ ]:

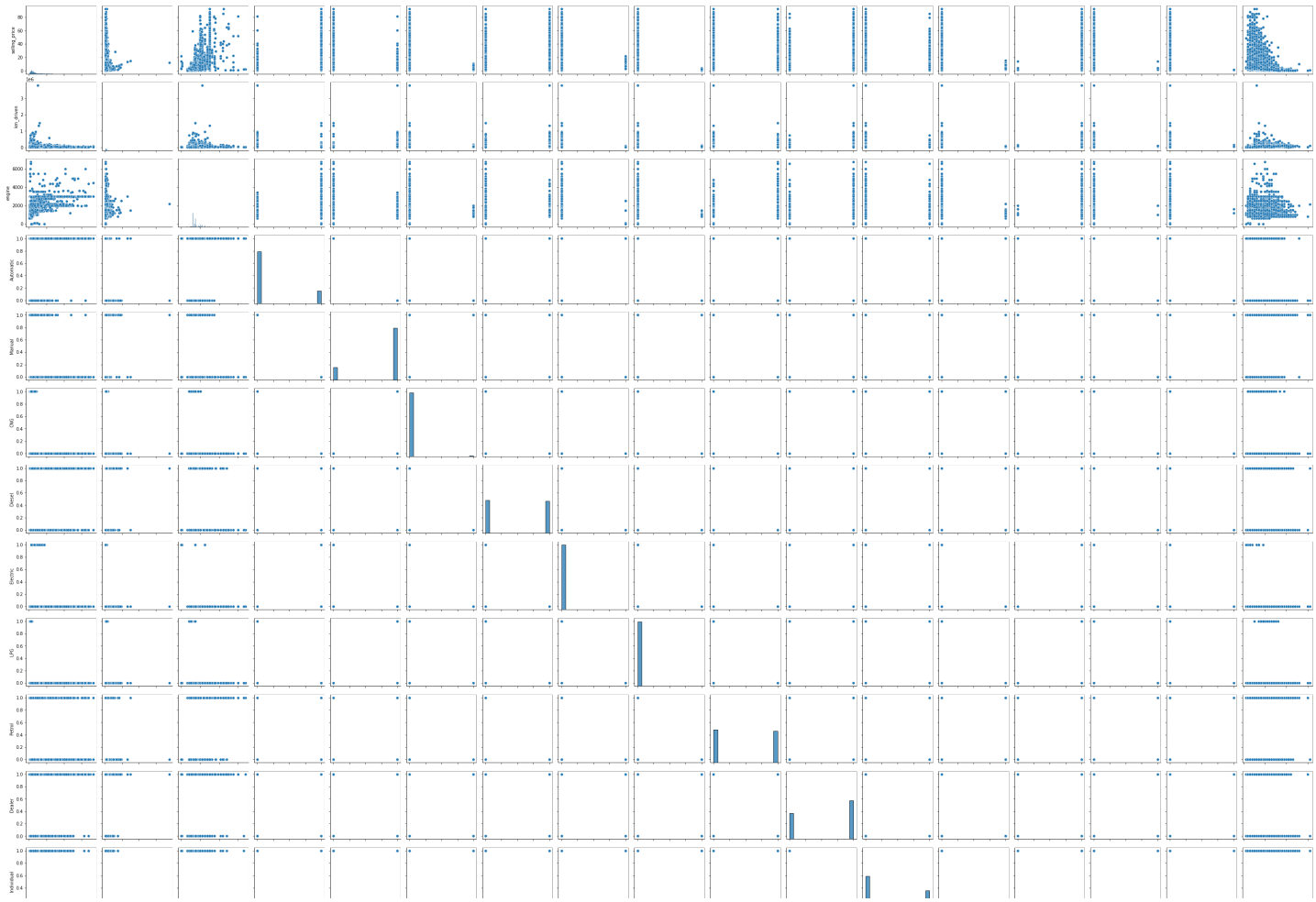
```
import seaborn as sns
```

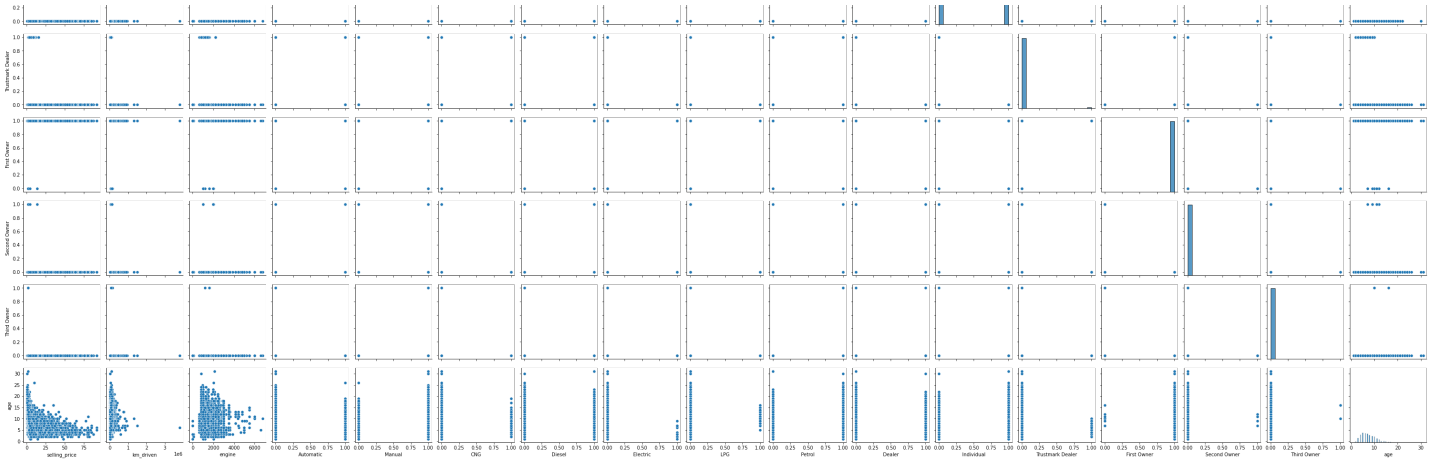
In [ ]:

```
## lets make a pairplot to check the various type of co-relations that exist in our data sets
sns.pairplot(newdf)
```

Out[ ]:

```
<seaborn.axisgrid.PairGrid at 0x7f5ac39cb3d0>
```



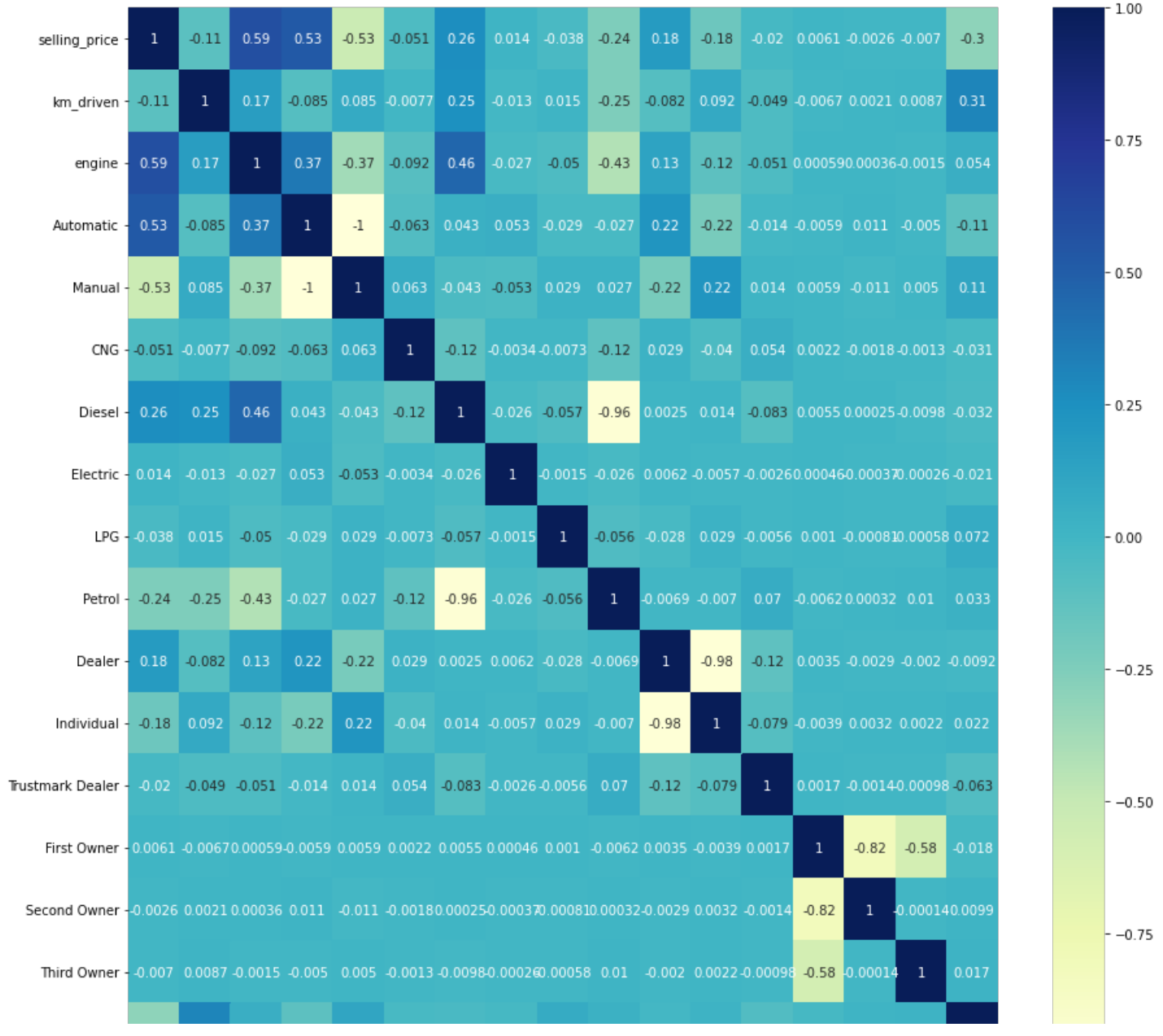


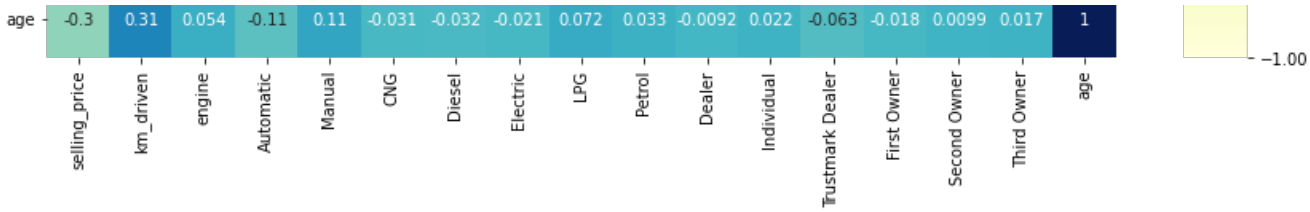
In [ ]:

```
import matplotlib.pyplot as plt
%matplotlib inline
```

In [ ]:

```
## it seems little tough to visulaise so many pair plots, now lets try to visualise it th
rough heat map
corrmat=newdf.corr()
top_corr_features=corrmat.index
plt.figure(figsize=(15,15))
## heatmap code https://seaborn.pydata.org/generated/seaborn.heatmap.html
g=sns.heatmap(newdf[top_corr_features].corr(),annot=True,cmap="YlGnBu")
```





If we observe the heat map carefully, we see many interesting findings

selling price which is our target variable is highly correlated with engine capacity , automatic transmission, and diesel vehicle

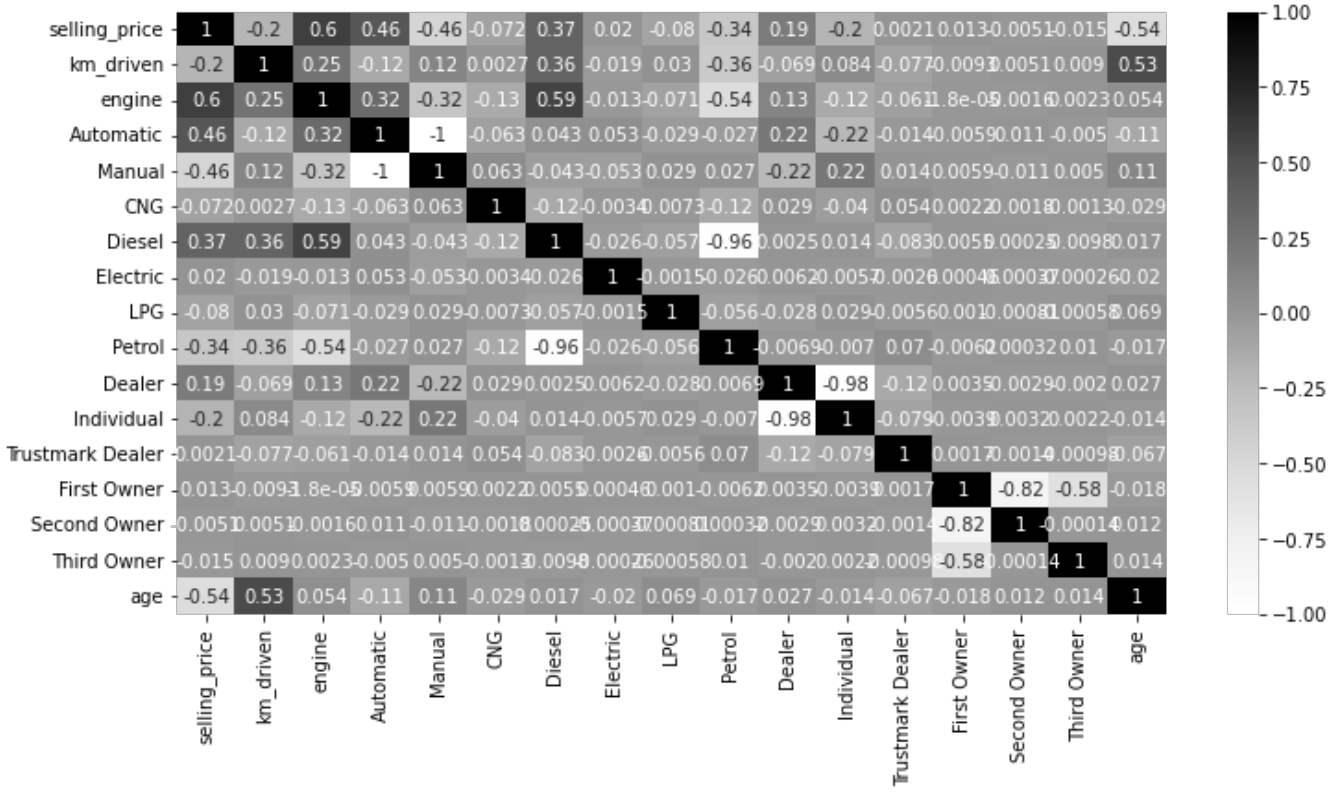
means that these features impact the selling price of vehicle positively, an automatic vehicle fetches better price than a manual, similarly a higher capacity engine is worth more in the market, and diesel vehicle will fetch a better price than its petrol counterpart

In [ ]:

```
## lets once check the spearman correlation
plt.figure(figsize=(12,6))
sns.heatmap(newdf.corr(method='spearman'),annot=True, cmap='Greys')
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f5ab76ae9d0>
```



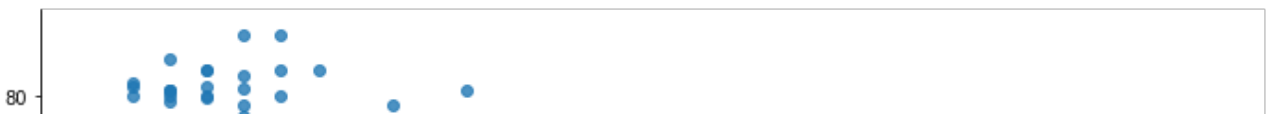
Here we see that a strong negative correlation is between selling price and killometers driven by a car, age of the vehicle, and transmission as selling price of a manual vehicle is prety less,similary a higher mileage car is fetching less price, and same is the case with age, higher the age of vehicle, lesser is its price. we should try to see these relations with selling price more graphically to get more clarity

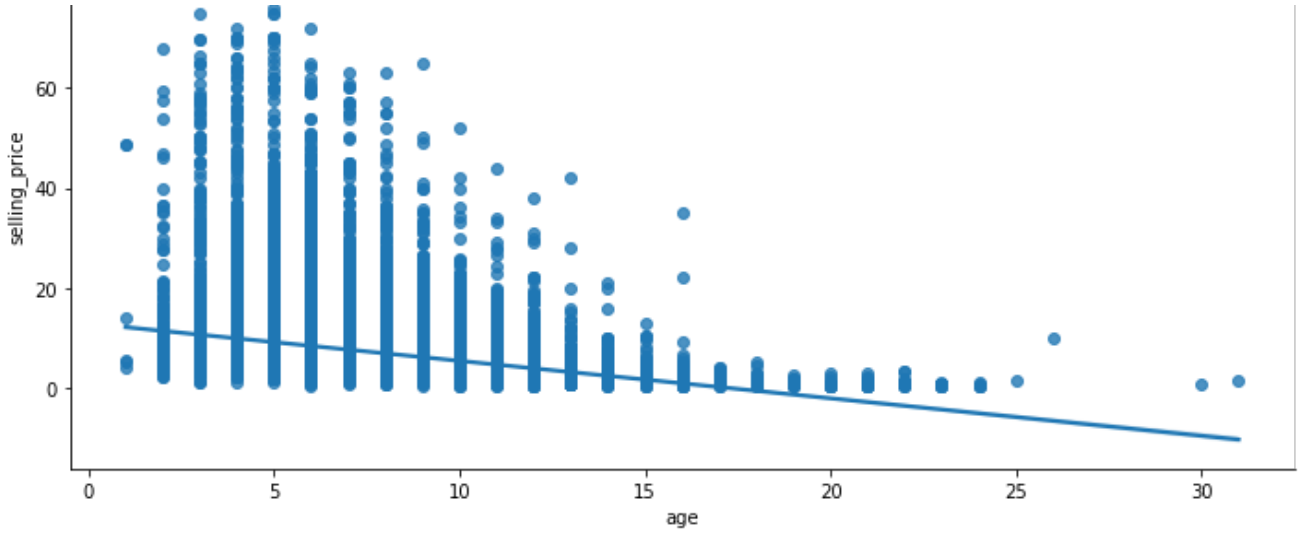
In [ ]:

```
sns.lmplot(x='age',y='selling_price',data=newdf,aspect=2,height=5)
```

Out[ ]:

```
<seaborn.axisgrid.FacetGrid at 0x7f5ab5aa5b10>
```





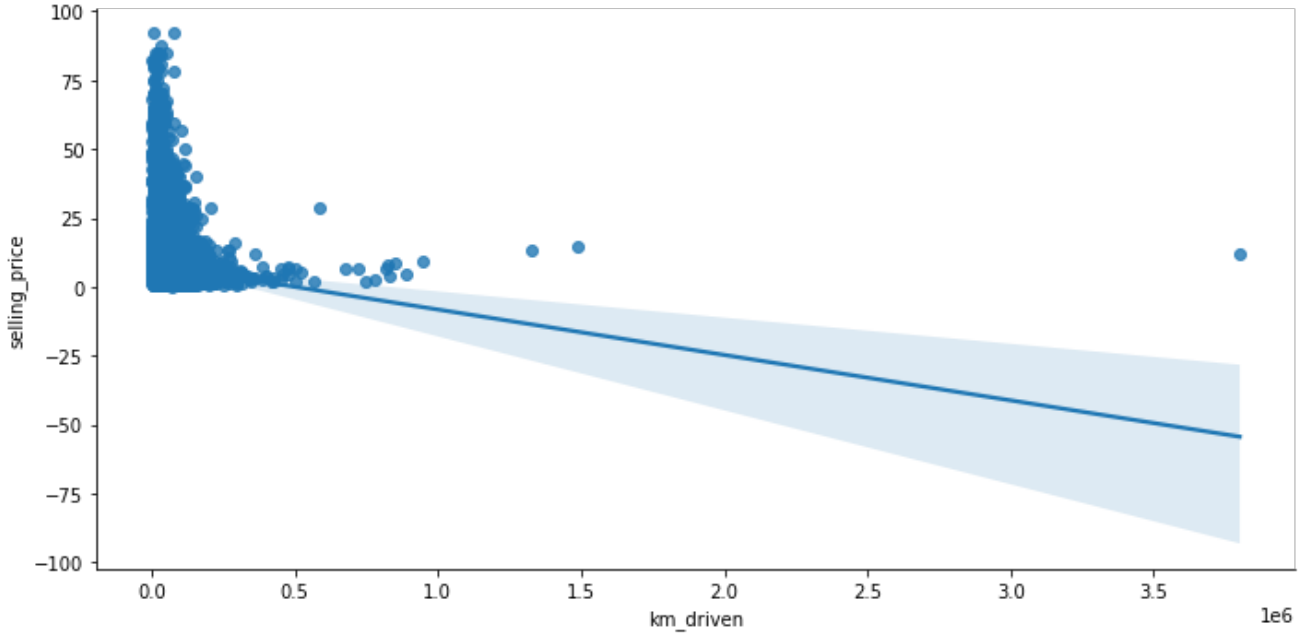
We can clearly see an inverse relation of age with selling price, as selling price keeps reducing as age is increasing

In [ ]:

```
sns.lmplot(x='km_driven',y='selling_price',data=newdf,aspect=2,height=5)
```

Out[ ]:

```
<seaborn.axisgrid.FacetGrid at 0x7f5ab5a1a690>
```



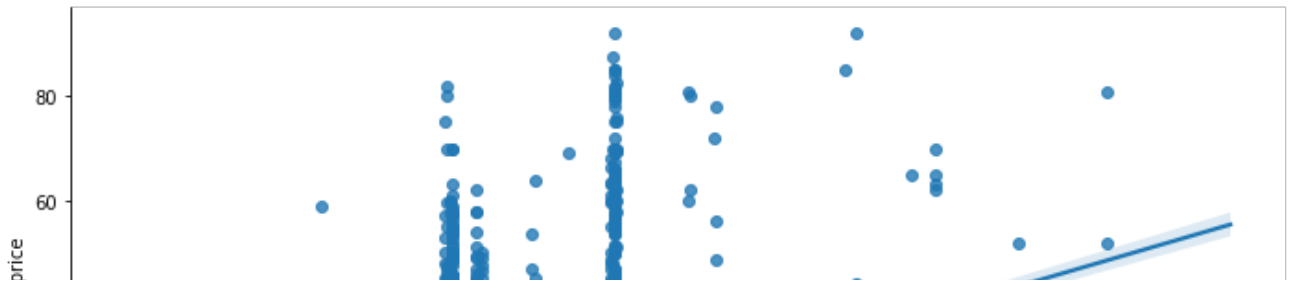
Same trend can be observed here as well but not very significant, age feature has a much bigger impact on price than mileage

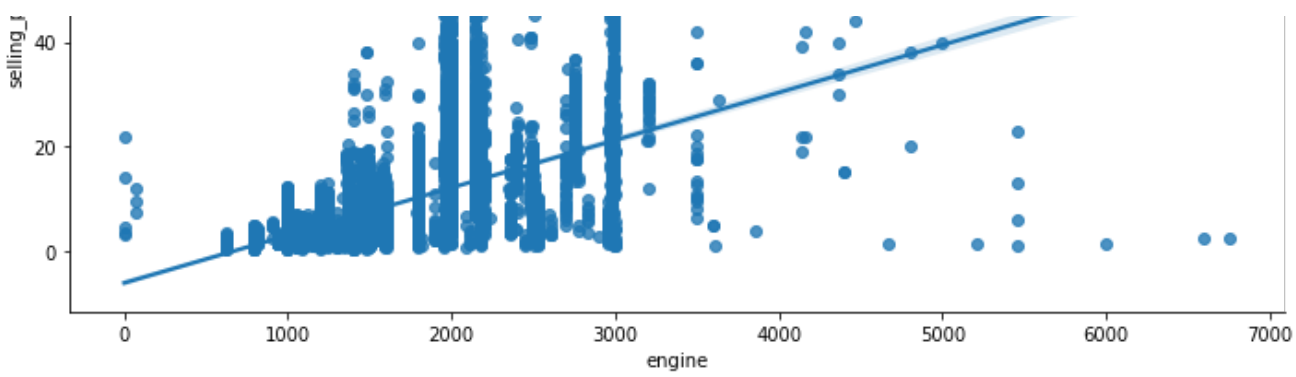
In [ ]:

```
sns.lmplot(x='engine',y='selling_price',data=newdf,aspect=2,height=5)
```

Out[ ]:

```
<seaborn.axisgrid.FacetGrid at 0x7f5ab59bb1d0>
```





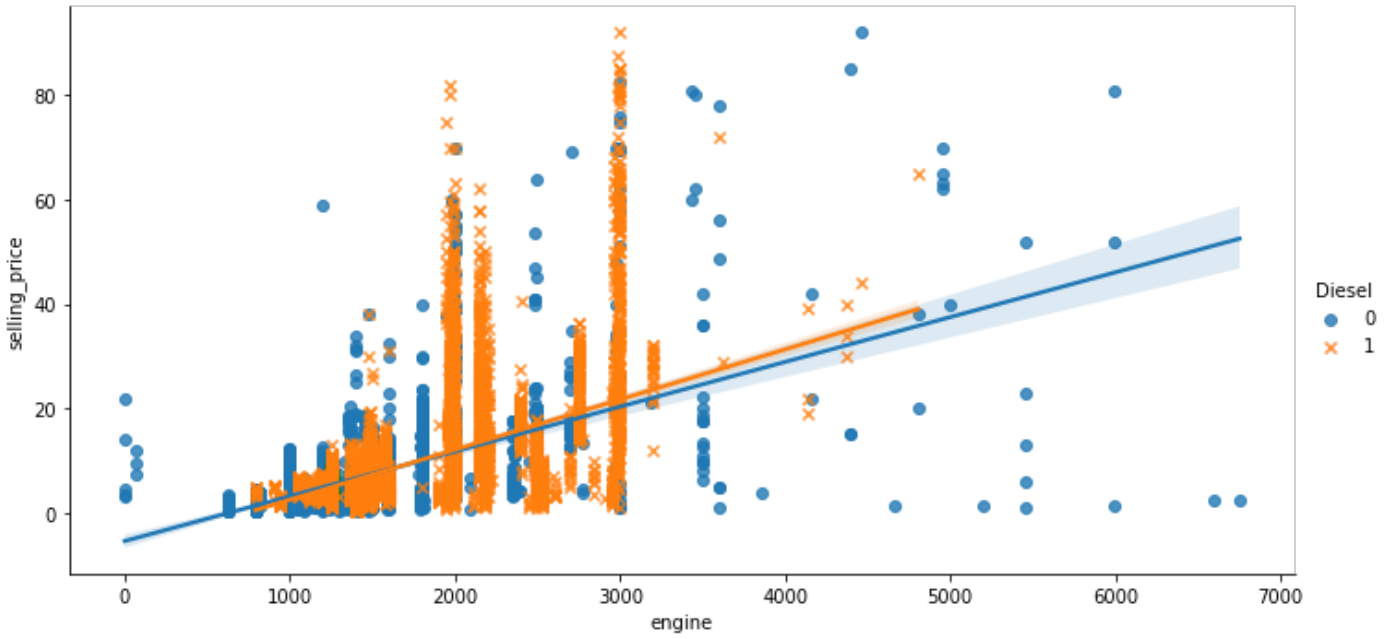
Engine capacity directly impacts the price here, but after 3000cc it does not have much impact, as above 3000 cc its the sports car territory, where the customers are limited and are not bothered about the engine capacity of their sports car

In [ ]:

```
sns.lmplot(x='engine',y='selling_price',data=newdf,aspect=2,height=5,hue='Diesel',markers=['o','x'])
```

Out[ ]:

```
<seaborn.axisgrid.FacetGrid at 0x7f5ab59841d0>
```



Here we can clearly see that a high capacity diesel engine has been in much more demand in the used car market which effects the selling price positively

In [ ]:

```
newdf.head(5)
```

Out[ ]:

	selling_price	km_driven	engine	Name	Automatic	Manual	CNG	Diesel	Electric	LPG	Petrol	Dealer	Individual	Trus
0	1.20	120000.0	796.0	MARUTI	0	1	0	0	0	0	1	0	1	
1	5.50	20000.0	1197.0	HYUNDAI	0	1	0	0	0	0	1	0	1	
2	2.15	60000.0	1197.0	HYUNDAI	0	1	0	0	0	0	1	0	1	
3	2.26	37000.0	998.0	MARUTI	0	1	0	0	0	0	1	0	1	
4	5.70	30000.0	1498.0	FORD	0	1	0	1	0	0	0	1	0	

In [ ]:

```
newdf.drop(['Name'], axis=1, inplace=True)
```

In [ ]:

```
## lets split our data into dependent and independent features
X=newdf.iloc[:,1:]
y=newdf.iloc[:,0]
```

In [ ]:

```
X.head()
```

Out[ ]:

	km_driven	engine	Automatic	Manual	CNG	Diesel	Electric	LPG	Petrol	Dealer	Individual	Trustmark Dealer	First Owner	Second Owner
0	120000.0	796.0	0	1	0	0	0	0	1	0	1	0	1	0
1	20000.0	1197.0	0	1	0	0	0	0	1	0	1	0	1	0
2	60000.0	1197.0	0	1	0	0	0	0	1	0	1	0	1	0
3	37000.0	998.0	0	1	0	0	0	0	1	0	1	0	1	0
4	30000.0	1498.0	0	1	0	1	0	0	0	1	0	0	1	0

In [ ]:

```
y.head()
```

Out[ ]:

```
0    1.20
1    5.50
2    2.15
3    2.26
4    5.70
Name: selling_price, dtype: float64
```

In [ ]:

```
## Feature importance
from sklearn.ensemble import ExtraTreesRegressor
model=ExtraTreesRegressor()
model.fit(X,y)
```

Out[ ]:

```
ExtraTreesRegressor()
```

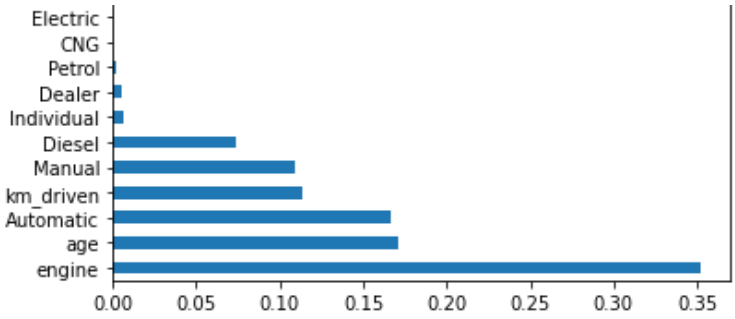
In [ ]:

```
print(model.feature_importances_)
```

```
[1.13794718e-01 3.52190498e-01 1.66370867e-01 1.08750717e-01
 9.15893809e-05 7.42840858e-02 8.89570007e-05 1.63545154e-05
 1.62191564e-03 5.44187898e-03 6.05981179e-03 5.85486260e-05
 7.24606477e-06 7.13546196e-06 4.63612109e-07 1.71215213e-01]
```

In [ ]:

```
## plotting graph for the feature importance
feat_importances = pd.Series(model.feature_importances_, index=X.columns)
feat_importances.nlargest(16).plot(kind='barh')
plt.show()
```



Well we can clearly see in the graph that the most important 6 features are engine, age, manual or automatic, km driven, and diesel vehicle, so rest all the features we basically dont need in our model

In [ ]:

```
newdf.drop(['Dealer','Individual','Petrol','Electric','CNG','Trustmark Dealer','LPG','Second Owner','First Owner','Third Owner'], axis=1, inplace=True)
```

In [ ]:

```
newdf.head()
```

Out[ ]:

	selling_price	km_driven	engine	Automatic	Manual	Diesel	age
0	1.20	120000.0	796.0	0	1	0	10.0
1	5.50	20000.0	1197.0	0	1	0	6.0
2	2.15	60000.0	1197.0	0	1	0	12.0
3	2.26	37000.0	998.0	0	1	0	10.0
4	5.70	30000.0	1498.0	0	1	1	7.0

In [ ]:

```
X=newdf.iloc[:,1:]
y=newdf.iloc[:,0]
```

In [ ]:

```
X.head()
## now we have all the relevant features
```

Out[ ]:

	km_driven	engine	Automatic	Manual	Diesel	age
0	120000.0	796.0	0	1	0	10.0
1	20000.0	1197.0	0	1	0	6.0
2	60000.0	1197.0	0	1	0	12.0
3	37000.0	998.0	0	1	0	10.0
4	30000.0	1498.0	0	1	1	7.0

In [ ]:

```
y.head()
```

Out[ ]:

```
0    1.20
1    5.50
2    2.15
3    2.26
4    5.70
Name: selling_price, dtype: float64
```

In [ ]:

```
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.25,random_state=42)
```

We will try different regression techniques to check the accuracy of each one. The technique which best fits the data , we will implement it from scratch

1: Linear Regression

In [ ]:

```
from sklearn.linear_model import LinearRegression
lr = LinearRegression()
lr.fit(X_train,y_train)
```

Out[ ]:

```
LinearRegression()
```

In [ ]:

```
## lets check cv score
import numpy as np
from sklearn.model_selection import cross_val_score
scores = cross_val_score(lr,X_train,y_train,cv=5,scoring='r2')
print('CV Mean: ', np.mean(scores))
print('STD: ', np.std(scores))
```

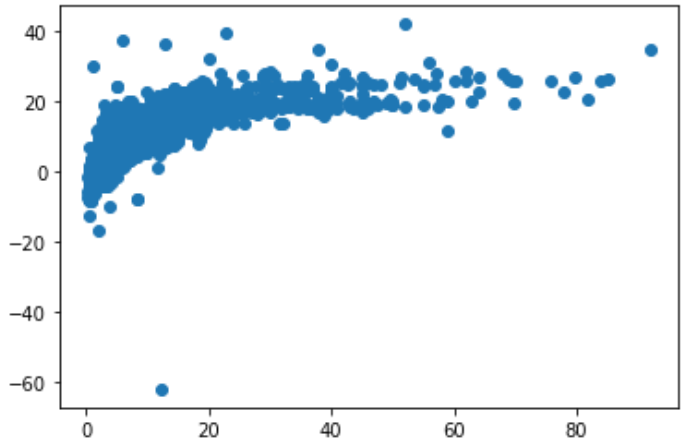
```
CV Mean:  0.5499879274293252
STD:  0.02098550087703452
```

In [ ]:

```
h = lr.predict(X_test)
plt.scatter(y_test,h)
```

Out[ ]:

```
<matplotlib.collections.PathCollection at 0x7f5ab200c710>
```



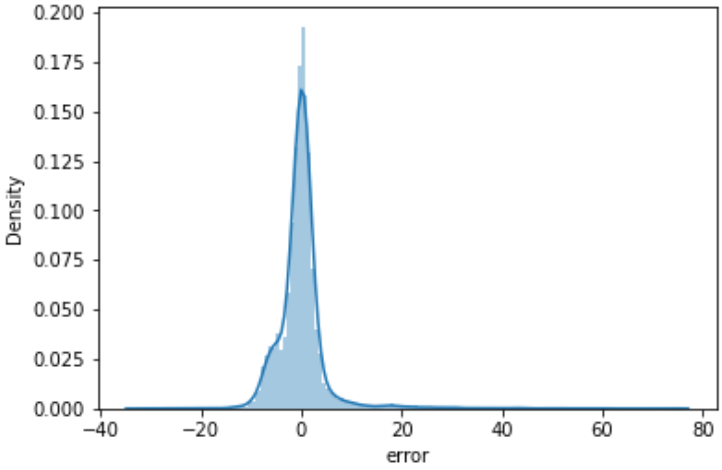
In [ ]:

```
sns.distplot((y_test-h),bins=150,axlabel = "error")
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/distributions.py:2619: FutureWarning: `dis
tplot` is a deprecated function and will be removed in a future version. Please adapt you
r code to use either `displot` (a figure-level function with similar flexibility) or `his
tplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f5ab1f81550>
```



In [ ]:

```
from sklearn import metrics

print('MAE:', metrics.mean_absolute_error(y_test, h))
print('MSE:', metrics.mean_squared_error(y_test, h))
print('RMSE:', np.sqrt(metrics.mean_squared_error(y_test, h)))
print('Model Accuracy:',lr.score(X_test, y_test)*100,'%')
```

```
MAE: 2.8685571875969864
MSE: 30.118231253963845
RMSE: 5.488007949517187
Model Accuracy: 53.81545830886483 %
```

Not getting Good results with linear regression

2: Random Forest

In [ ]:

```
from sklearn.ensemble import RandomForestRegressor
rf_random=RandomForestRegressor()
```

In [ ]:

```
## Hyperparameters
## we will circle through different parameters for our trees
n_estimators = [int(x) for x in np.linspace(start=100,stop=1300, num=11)]
print(n_estimators)
```

```
[100, 220, 340, 460, 580, 700, 820, 940, 1060, 1180, 1300]
```

In [ ]:

```
rf_random.fit(X_train,y_train)
```

Out[ ]:

```
RandomForestRegressor()
```

In [ ]:

```
predictions=rf_random.predict(X_test)
```

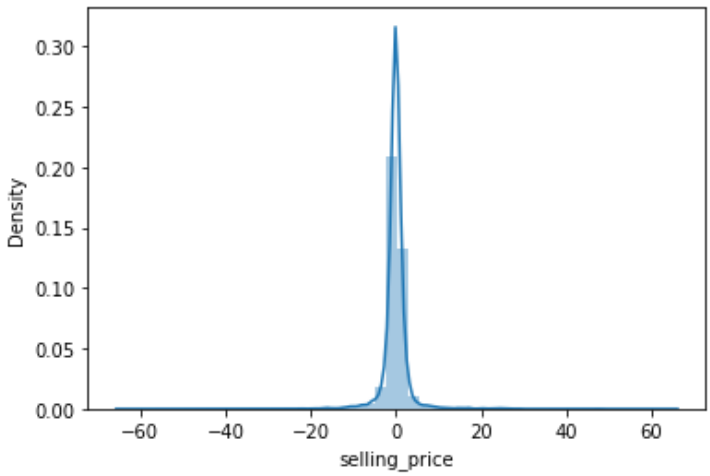
In [ ]:

```
sns.distplot(y_test-predictions)
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/distributions.py:2619: FutureWarning: `dis
tplot` is a deprecated function and will be removed in a future version. Please adapt you
r code to use either `displot` (a figure-level function with similar flexibility) or `his
tplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f5ab58e7610>
```

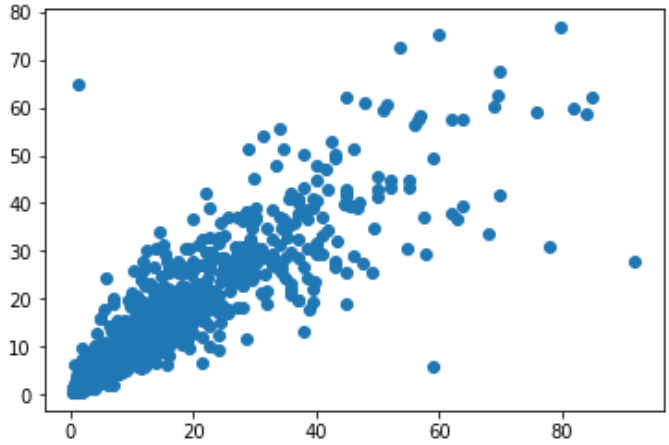


In [ ]:

```
plt.scatter(y_test,predictions)
```

Out[ ]:

```
<matplotlib.collections.PathCollection at 0x7f5ab1c0abd0>
```



In [ ]:

```
df=pd.DataFrame({'Actual':y_test, 'Predicted':predictions})
df
```

Out[ ]:

	Actual	Predicted
4022	2.70	1.782600
2092	9.51	10.151500
4780	7.50	6.746733
10652	19.00	11.397351
15472	12.75	14.304767
...	...	...
13979	6.75	8.405800
19653	1.20	1.287361
5355	2.91	2.700837
19501	6.51	6.528367
10204	2.20	1.679025

4995 rows × 2 columns

In [ ]:

```
from sklearn import metrics
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, predictions))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, predictions))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, predictions)
))
```

```
Mean Absolute Error: 1.4763877746975786
Mean Squared Error: 11.20863114444786
Root Mean Squared Error: 3.3479293816399203
```

In [ ]:

```
# Calculate the absolute errors
errors = abs(predictions - y_test)
# Print out the mean absolute error (mae)
print('Mean Absolute Error:', round(np.mean(errors), 2), 'degrees.')

# Calculate mean absolute percentage error (MAPE)
mape = 100 * (errors / y_test)
# Calculate and display accuracy
accuracy = 100 - np.mean(mape)
print('Accuracy:', round(accuracy, 2), '%.')
```

```
Mean Absolute Error: 1.48 degrees.
Accuracy: 82.34 %.
```

We have reached an approximate 80 percent accuracy here, in the next phase we will try to fine tune and try some other models to see if we can improve on our accuracy

3: XGBoost Regression

In [ ]:

```
from xgboost import XGBRegressor
```

In [ ]:

```
xg_reg=XGBRegressor()
```

In [ ]:

```
xg_reg.fit(X_train,y_train)
```

```
[10:48:46] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now dep
recated in favor of reg:squarederror.
```

Out[ ]:

```
XGBRegressor()
```

In [ ]:

```
y_pred=xg_reg.predict(X_test)
```

In [ ]:

```
y_pred
```

Out[ ]:

```
array([ 2.6788568, 10.703831 ,  6.6694684, ...,  2.2996194,  6.136638 ,
        1.4247425], dtype=float32)
```

In [ ]:

```
difference = y_test - y_pred
```

In [ ]:

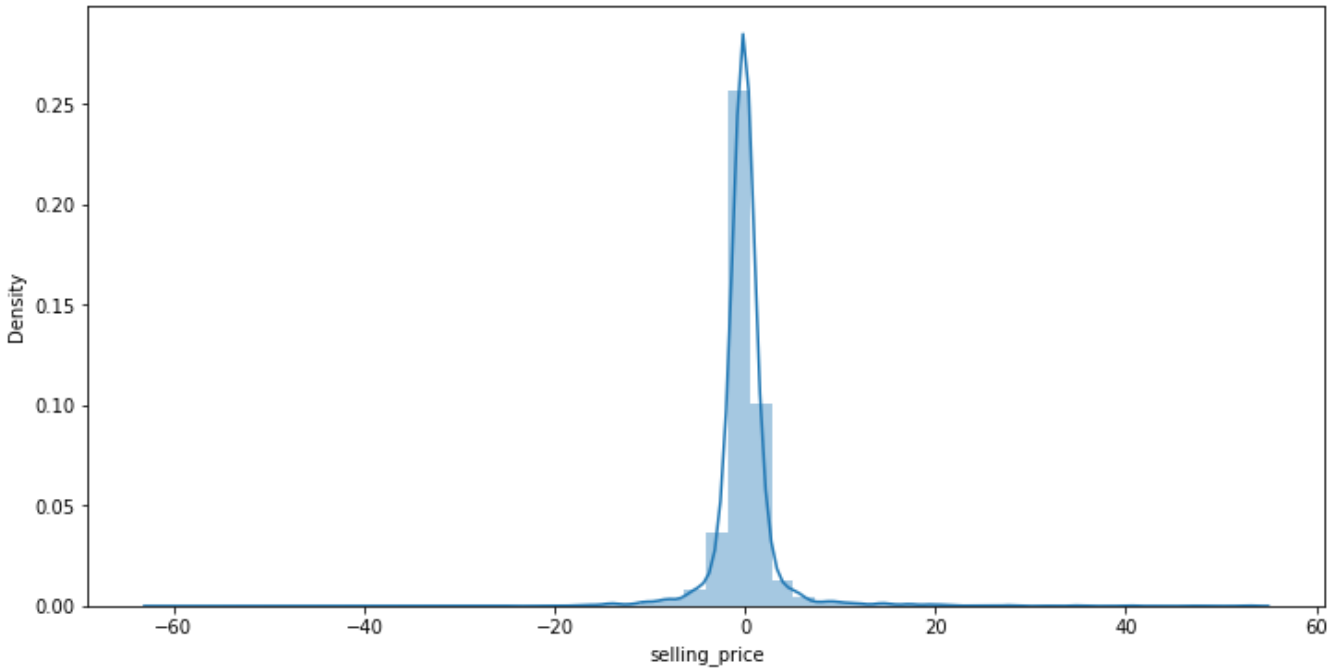
In [ ]:

```
plt.figure(figsize= (12,6))
sns.distplot(difference)
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/distributions.py:2619: FutureWarning: `dis
tplot` is a deprecated function and will be removed in a future version. Please adapt you
r code to use either `displot` (a figure-level function with similar flexibility) or `his
tplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f5aa8d763d0>
```



In [ ]:

```
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
```

```
Mean Absolute Error: 1.6326354949191095
Mean Squared Error: 12.571552425069207
Root Mean Squared Error: 3.5456385073875207
```

In [ ]:

```
from sklearn.model_selection import cross_val_score
```

In [ ]:

```
model= XGBRegressor(objective="reg:squarederror")
```

In [ ]:

```
scores = cross_val_score(model, X, y,
                         scoring = "neg_mean_squared_error",
                         cv=10)
```

In [ ]:

```
scores
```

Out[ ]:

```
array([-16.72258769, -12.879534  , -13.15327242,  -7.33598788,
        -9.43653097,  -6.11804584, -23.55500705, -15.30221987,
       -11.56557937,  -9.98103339])
```

In [ ]:

```
rmse=np.sqrt(-scores)
```

In [ ]:

```
print(rmse)
# with CV we rached a minimum RMSE of 2.47 , earlier we had an rmse of 3.3, so CV helped us reducing it
```

```
[4.08932607 3.58880677 3.62674405 2.70850288 3.07189371 2.47346838
 4.85335009 3.91180519 3.4008204  3.15927735]
```

In [ ]:

```
xg_reg.score(X_test,y_test)*100
#result on testing
```

Out[ ]:

```
80.72226147006936
```

In [ ]:

```
df=pd.DataFrame({'Actual':y_test, 'Predicted':y_pred})
df
```

Out[ ]:

	Actual	Predicted
4022	2.70	2.678857
2092	9.51	10.703831
4780	7.50	6.669468
10652	19.00	14.245301
15472	12.75	14.039997
...	...	...
13979	6.75	7.060275
19653	1.20	2.486934
5355	2.91	2.299619
19501	6.51	6.136638
10204	2.20	1.424742

4995 rows × 2 columns

In [ ]:

```
xgberrors = abs(y_pred - y_test)
# Calculate mean absolute percentage error (MAPE)
mape_xgb = 100 * (xgberrors / y_test)
# Calculate and display accuracy
accuracy = 100 - np.mean(mape_xgb)
print('Accuracy:', round(accuracy, 2), '%.')
```

```
Accuracy: 74.16 %.
```

Both Random forest and XGB regressor gave us good results but random forest has been a comparitively better

In [ ]:

```

```