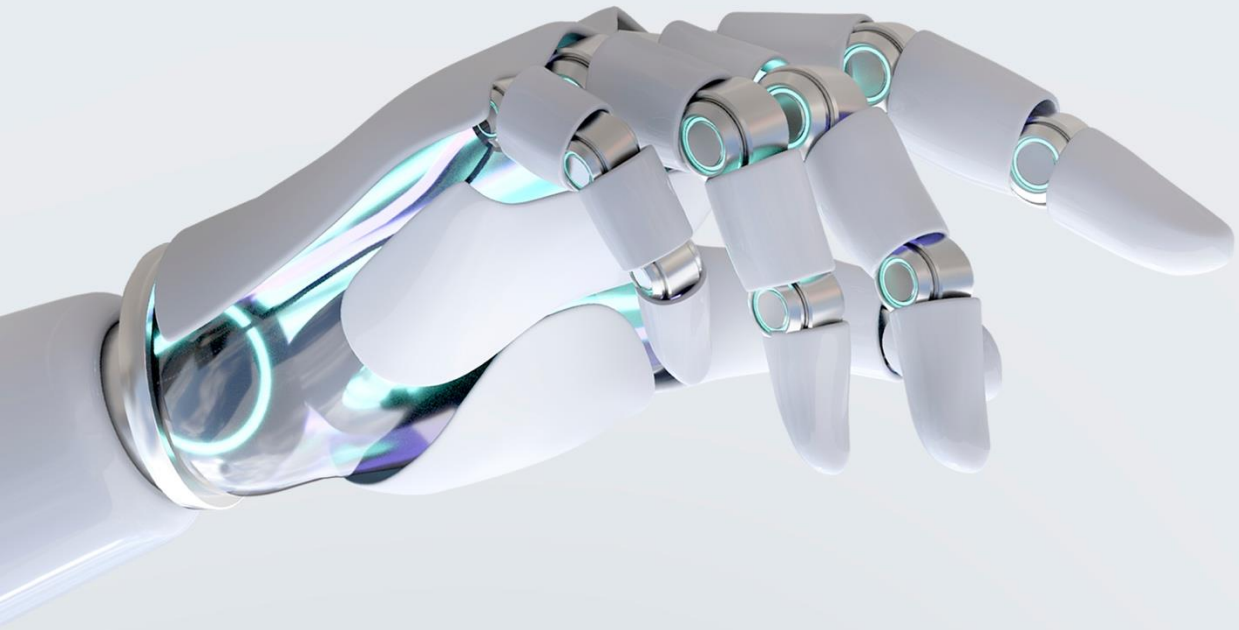


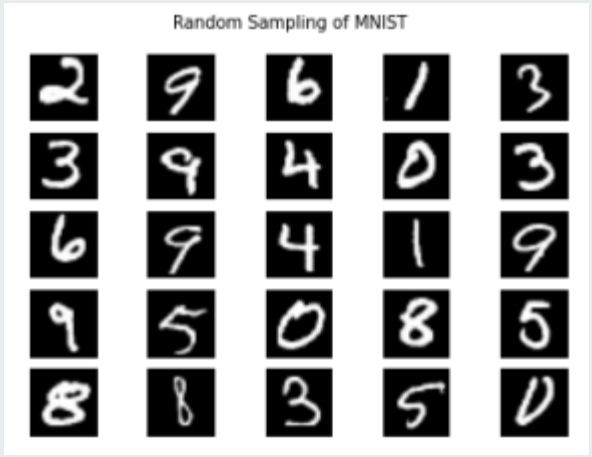
MACHINE LEARNING

Presented by

Sourav Ghosh
Senior Research Fellow
PI- Prof. Ranjit Thapa
Department of Physics
SRM University AP

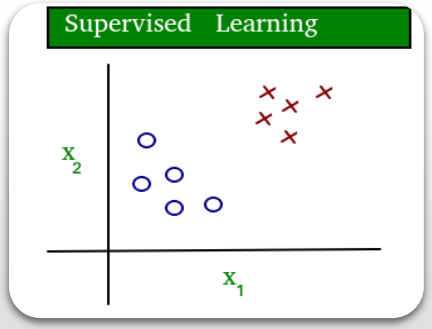
Machine Learning is the field of study that gives computers the capability to learn without being explicitly programmed.

Machine learning is a branch of artificial intelligence (AI) and computer science which focuses on the use of data and algorithms to imitate the way that humans learn, gradually improving its accuracy.



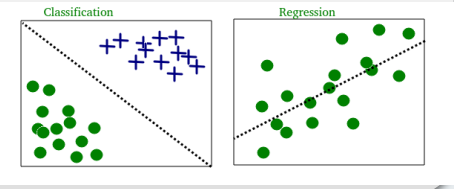
Ref – GeeksforGeeks, IBM, Google Images

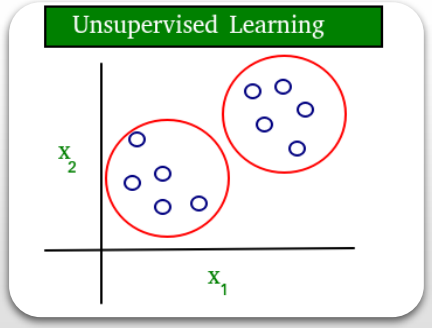
Types of machine learning problems



Supervised Learning

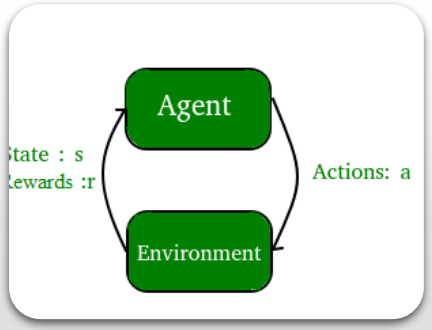
- The model or algorithm is presented with example inputs and their desired outputs and then finding patterns and connections between the input and the output. The goal is to learn a general rule that maps inputs to outputs.
- Applications-Image Classification, Market Prediction/Regression
- The data in supervised learning is labelled





Unsupervised Learning

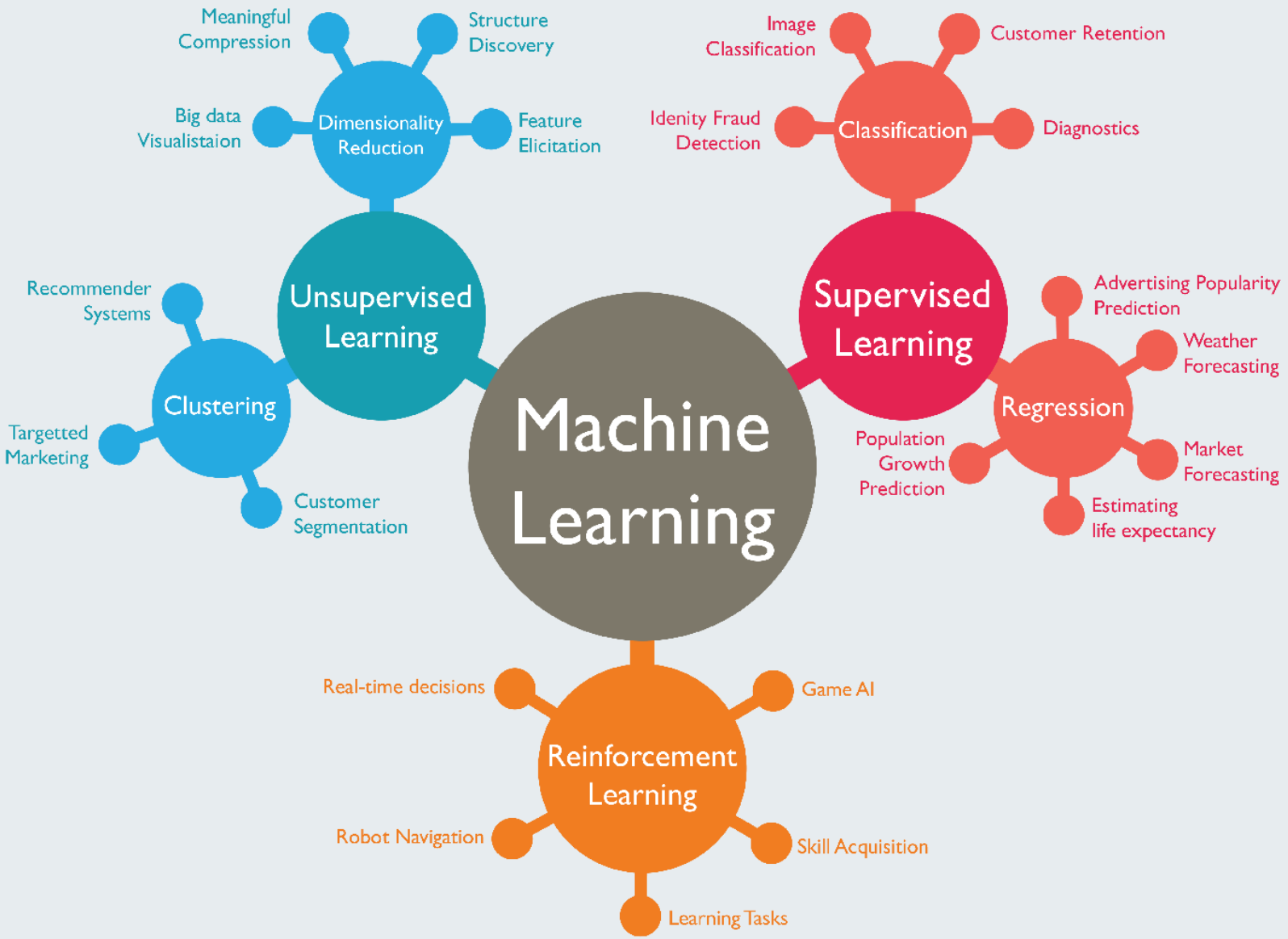
- No labels are given to the learning algorithm, leaving it on its own to find structure in its input.
- Applications- Clustering, High Dimension Visualization, Generative Models



Reinforcement Learning

- A computer program interacts with a dynamic environment in which it must perform a certain goal (such as driving a vehicle or playing a game against an opponent). The program is provided feedback in terms of rewards and punishments as it navigates its problem space.

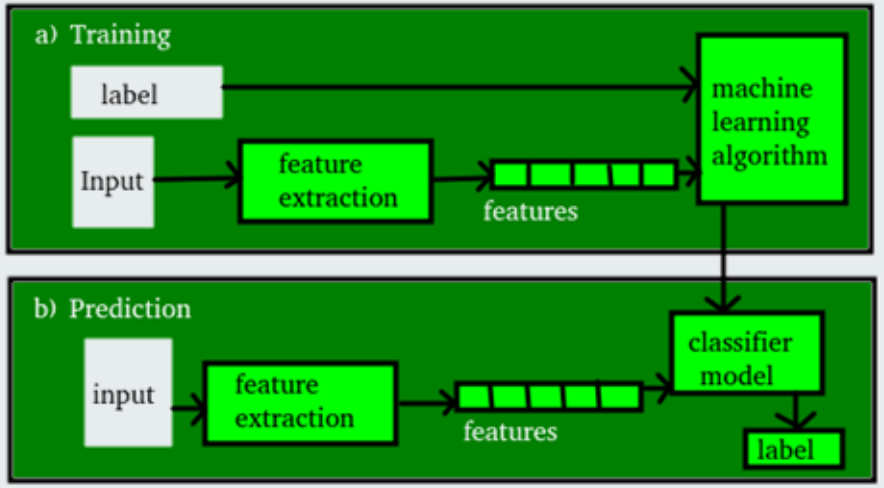
Reference – Geeks for Geeks



Reference – Google Images

Terminologies in Machine Learning

Model	• A model is a **specific representation** learned from data by applying some machine learning algorithm. A model is also called **hypothesis**.
Feature	• A feature is an individual measurable property of our data. A set of numeric features can be conveniently described by a **feature vector**. Feature vectors are fed as input to the model. For example, in order to predict a fruit, there may be features like color, smell, taste, **etc.**
Label	• A target variable or label is the value to be predicted by our model. For the fruit example discussed in the features section, the label with each set of input would be the name of the fruit like apple, orange, banana, etc..
Training	• The idea is to give a set of inputs(features) and it's expected outputs(labels), so after training, we will have a model (hypothesis) that will then map new data to one of the categories trained on.
Prediction	• Once our model is ready, it can be fed a set of inputs to which it will provide a predicted output(label). But make sure if the machine performs well on unseen data, then only we can say the machine performs well.



Reference – GeeksforGeeks

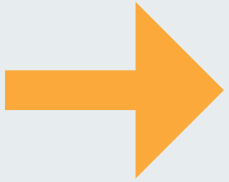
In most supervised machine learning tasks, best practice recommends to split your data into three independent sets: a **training set**, a **testing set**, and a **validation set**.

To learn why, let's pretend that we have a dataset of two types of pets:

Cats: 🐱 Dogs: 🐶

Each pet in our dataset has two features: **weight** and **fluffiness**.

Our goal is to identify and evaluate suitable models for classifying a given pet as either a cat or a dog. We'll use train/test/validations splits to do this!




Test


Train


Validation

Train, Test, and Validation Splits

The first step in our classification task is to randomly split our pets into three independent sets:

Training Set: The dataset that we feed our model to learn potential underlying patterns and relationships.

Validation Set: The dataset that we use to understand our model's performance across different model types and hyperparameter choices.

Test Set: The dataset that we use to approximate our model's unbiased accuracy in the wild.

The Training Set

The training set is the dataset that we employ to train our model. It is this dataset that our model uses to learn any underlying patterns or relationships that will enable making predictions later on.

The training set should be as representative as possible of the population that we are trying to model. Additionally, we need to be careful and ensure that it is as unbiased as possible, as any bias at this stage may be propagated downstream during inference.

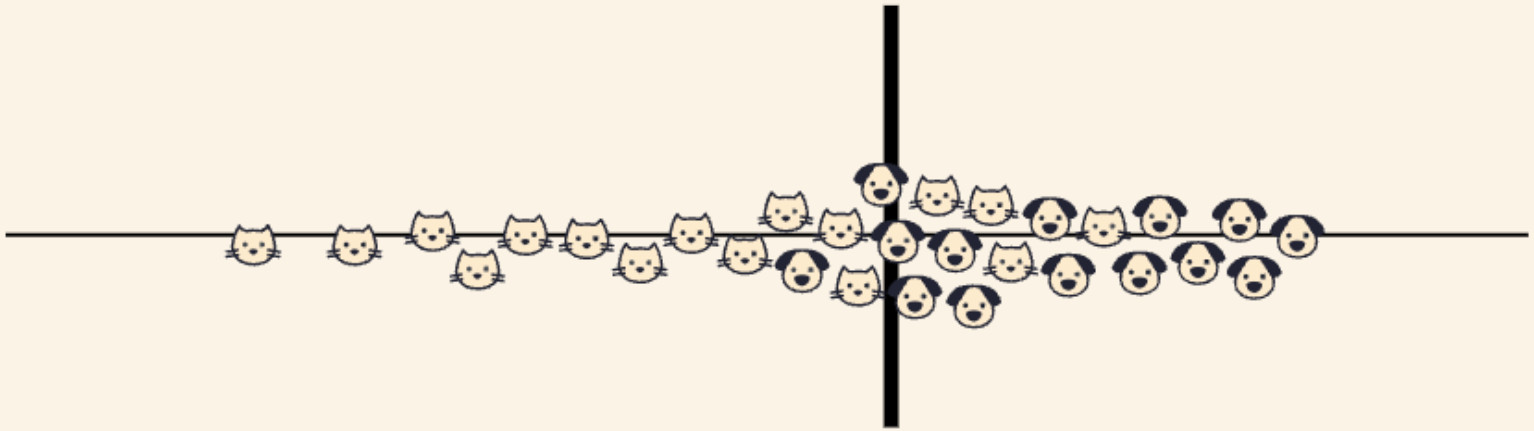
Building Our Model

Model Features: **None** **Weight** **Fluffiness** **Both**

Our goal (to determine whether a given pet is a cat or a dog) is a binary classification task, so we will use a simple but effective model appropriate for this task: **logistic regression**.

Logistic regression will learn a decision boundary to best separate the cats from dogs in our training data, using the selected feature (**None**, **Weight**, **Fluffiness**, or *both* **Weight** and **Fluffiness**).

Select the feature to visualize the corresponding logistic regression model's decision boundary. **Drag each animal in the training set to a new position to see how the boundary updates!**



Weight

The Validation Set

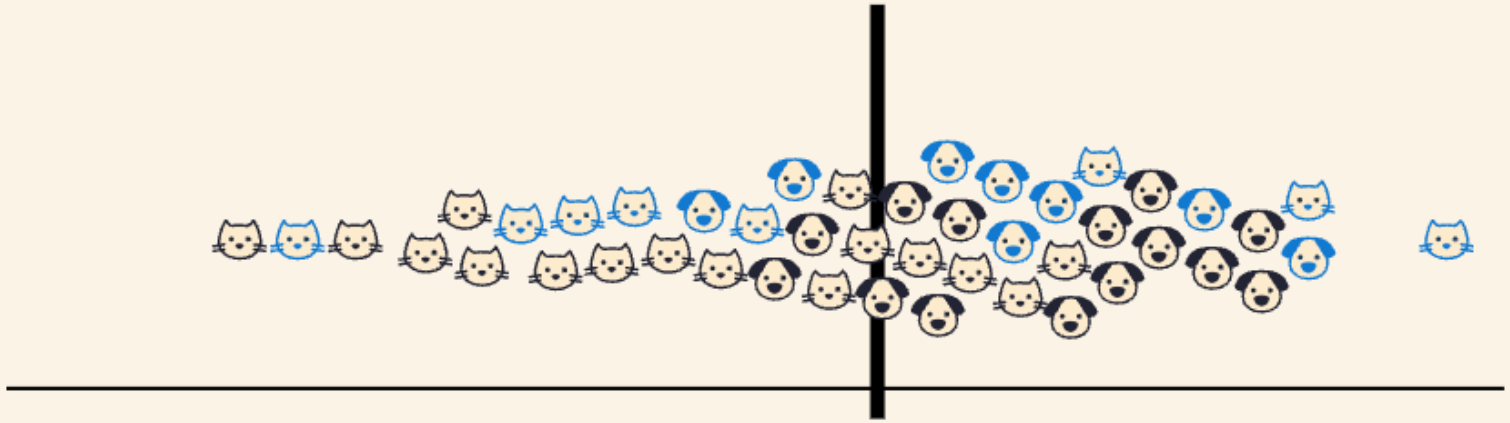
Model Features: **None** | **Weight** | **Fluffiness** | **Both**

We can build four different logistic regression models (one for each feature possibility), **how do we decide which model to select?**

We could compare the accuracy of each model on the training set, but if we use the same exact dataset for both training and tuning, the model will overfit and won't generalize well.

This is where the validation set comes in — it acts as an independent, unbiased dataset for comparing the performance of different algorithms trained on our training set.

Select a feature to view the model's performance on the validation set in the table below. **Drag the pets across the line to see how the model performance updates!**



Weight

dataset	feature	# cat right	# cat wrong	# dog right	# dog wrong	accuracy
validation	weight	5	3	6	2	68.8%

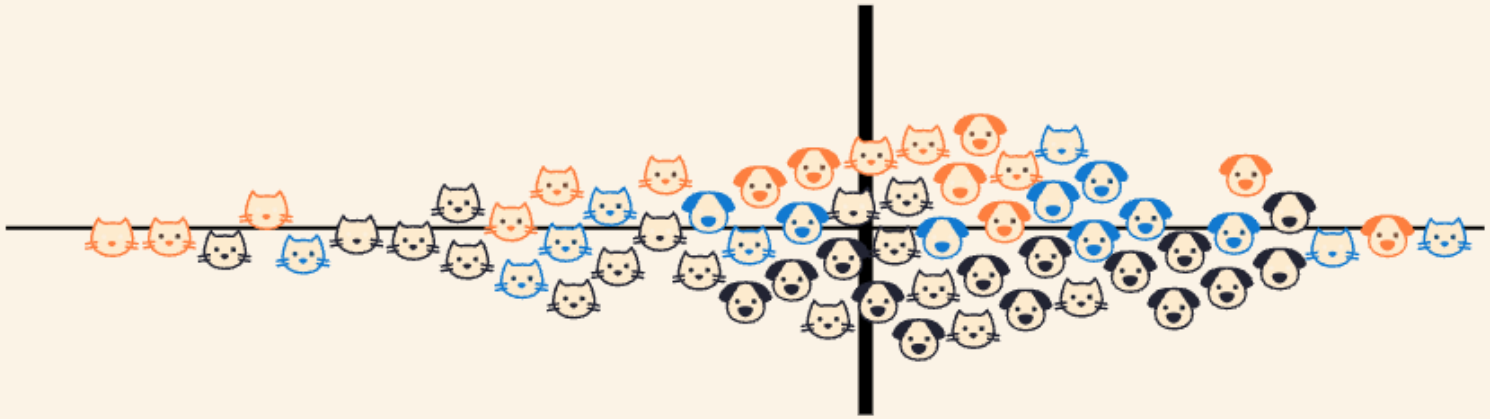
The Testing Set

Once we have used the validation set to determine the algorithm and parameter choices that we would like to use in production, the test set is used to approximate the models's true performance in the wild. It is the final step in evaluating our model's performance on unseen data.

We should never, under any circumstance, look at the test set's performance before selecting a model.

Peeking at our test set performance ahead of time is a form of overfitting, and will likely lead to unreliable performance expectations in production. It should only be checked as the final form of evaluation, after the validation set has been used to identify the best model.

Model Features: None | **Weight** | Fluffiness | Both



Weight

dataset	feature	# cat right	# cat wrong	# dog right	# dog wrong	accuracy
test	weight	7	2	5	2	75.0%
validation	weight	5	3	6	2	68.8%

Linear Regression

- Linear regression analysis is used to predict the value of a variable based on the value of another variable. The variable you want to predict is called the dependent variable. The variable you are using to predict the other variable's value is called the independent variable.

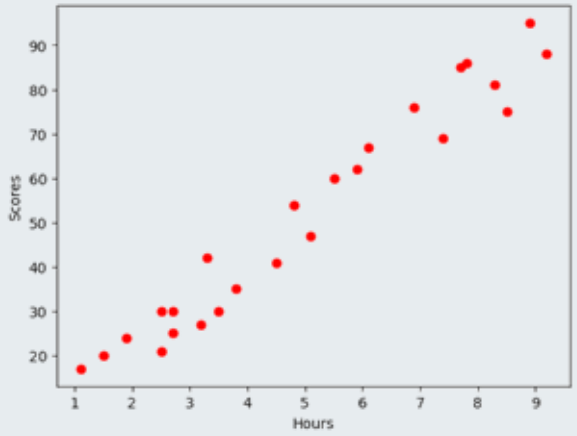
$$y = mx + c$$

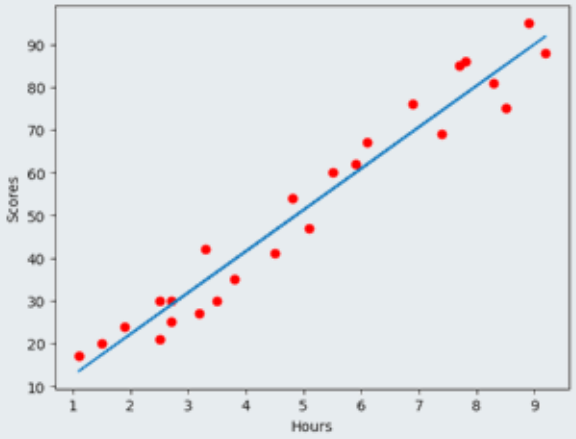
Dependent Variable

Slope

independent Variable

Intercept





c =2.82689235

m =9.68207815

$score$=9.68207815*$hours$+2.82689235

That's the heart of linear regression and an algorithm really only figures out the values of the slope and intercept. By modelling that linear relationship, our regression algorithm is also called a model.

But according to it many lines are possible. Why only one line ?

Reference – Google

Let's Be More Specific

Linear regression is a supervised algorithm[i] that learns to model a dependent variable, y, as a function of some independent variables (aka "features"), x_i, by finding a line (or surface) that best "fits" the data. In general, we assume y to be some number and each x_i can be basically anything. For example: predicting the price of a house using the number of rooms in that house (y: price, x_1: number of rooms) or predicting weight from height and age (y: weight, x_1: height, x_2: age).

In general, the equation for linear regression is

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_p x_p + \epsilon$$

where:

y: the dependent variable; the thing we are trying to predict.[i]

x_i: the independent variables: the features our model uses to model y.[i]

β_i: the coefficients (aka "weights") of our regression model. These are the foundations of our model. They are what our model "learns" during optimization.[i]

ϵ: the irreducible error in our model. A term that collects together all the unmodeled parts of our data.

Fitting a linear regression model is all about finding the set of cofficients that best model y as a function of our features. We may never know the true parameters for our model, but we can estimate them (more on this later). Once we've estimated these coefficients, $\hat{\beta}_i$, we predict future values, $\hat{y}$, as:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + ... + \hat{\beta}_p x_p$$

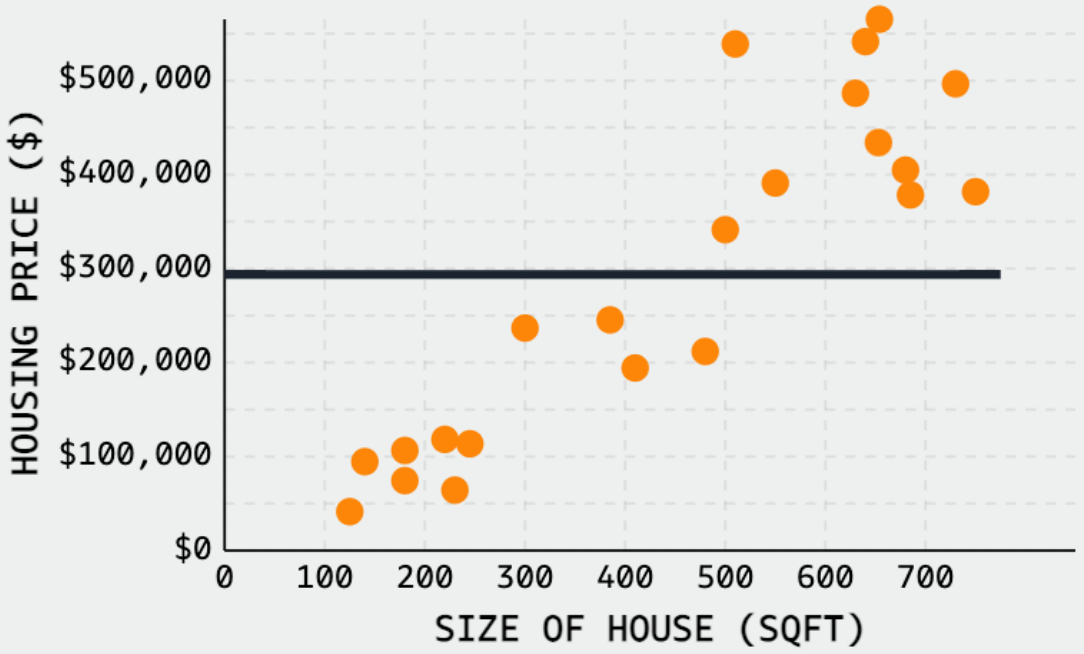
So predicting future values (often called inference), is as simple as plugging the values of our features x_i into our equation!

Let's fit a model to predict housing price ($) in San Diego, USA using the size of the house (in square-footage):

$$\text{house-price} = \hat{\beta}_1 * sqft + \hat{\beta}_0$$

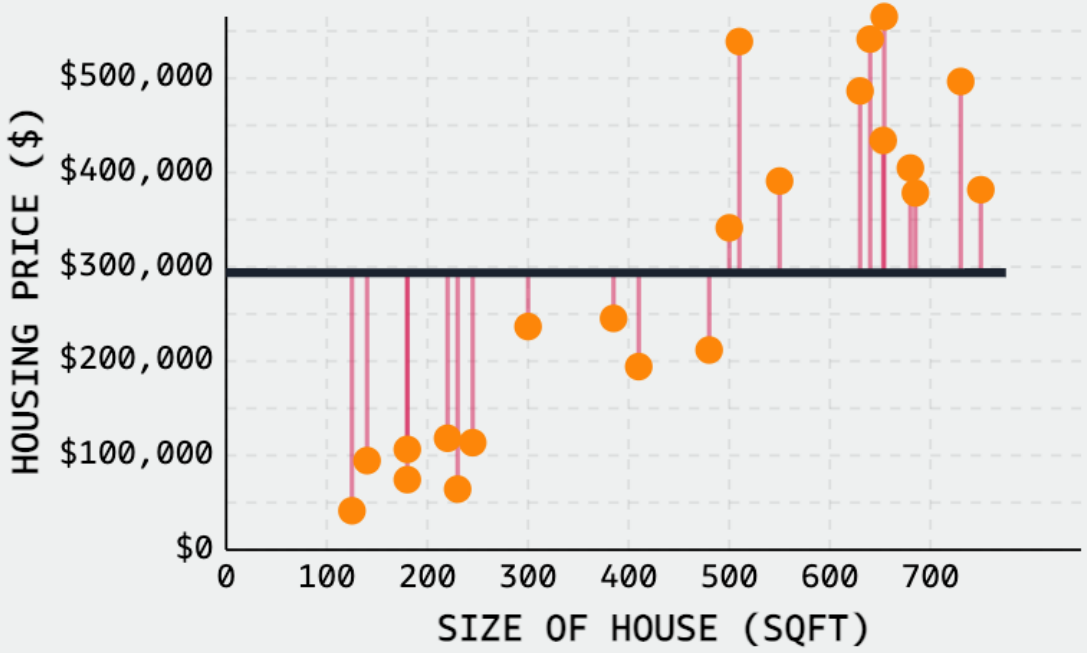
We'll start with a very simple model, predicting the price of each house to be just the average house price in our dataset, ~$290,000, ignoring the different sizes of each house:

$$\text{house-price} = 0 * sqft + 290000$$



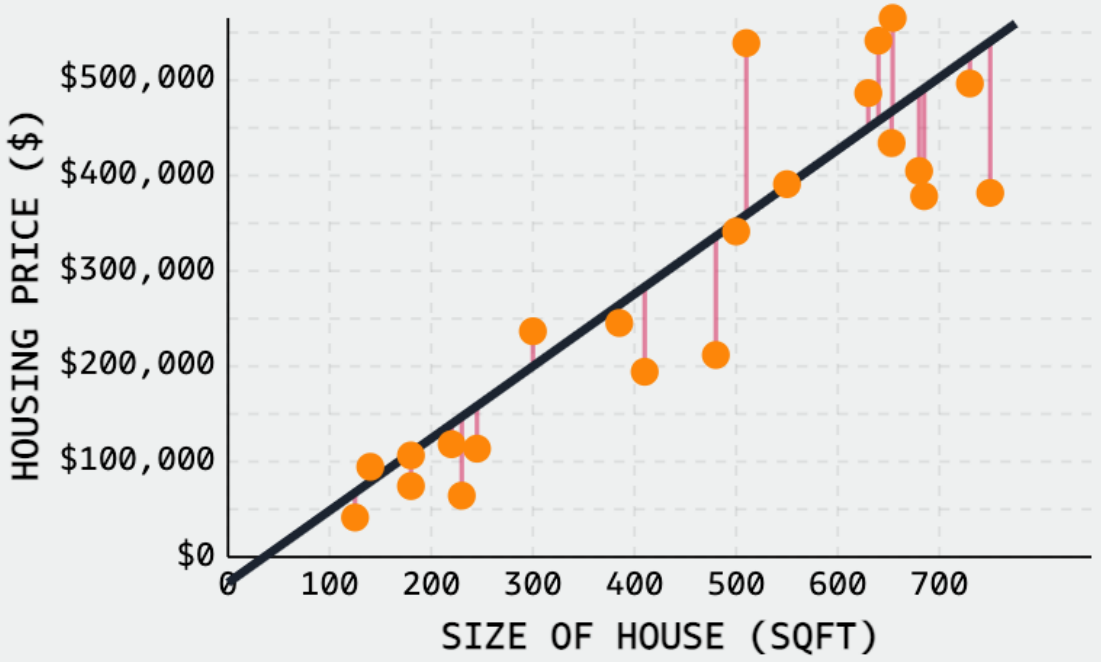
Of course we know this model is bad - the model doesn't fit the data well at all. But how can do quantify exactly *how* bad?

To evaluate our model's performance quantitatively, we plot the error of each observation directly. These errors, or **residuals**, measure the distance between each observation and the predicted value for that observation. We'll make use of these residuals later when we talk about evaluating regression models, but we can clearly see that our model has a lot of error.



The goal of linear regression is reducing this error such that we find a line/surface that 'best' fits our data. For our simple regression problem, that involves estimating the y-intercept and slope of our model, $\hat{\beta}_0$ and $\hat{\beta}_1$.

For our specific problem, the best fit line is shown. There's still error, sure, but the general pattern is captured well. As a result, we can be reasonably confident that if we plug in new values of square-footage, our predicted values of price would be reasonably accurate.



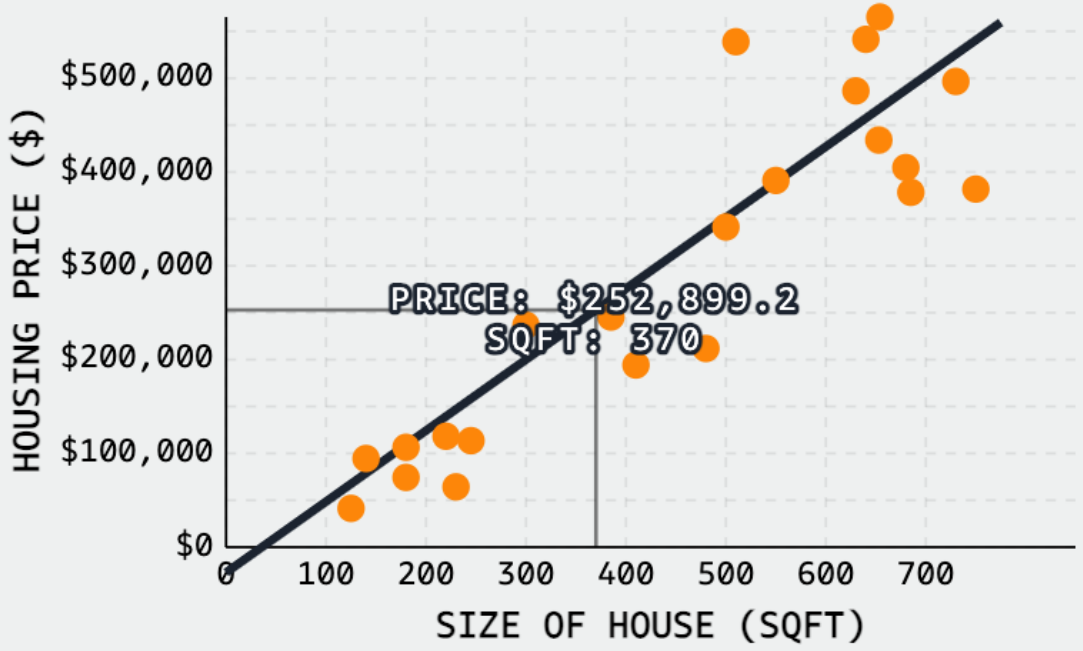
Once we've fit our model, predicting future values is super easy! We just plug in any x_i values into our equation!

For our simple model, that means plugging in a value for $sqft$ into our model:

$sqft$ Value: 370

$$\hat{y} = 756.9 * 370 - 27153.8$$
$$\hat{y} = 252899$$

Thus, our model predicts a house that is 370 square-feet will cost $252,899.



Model Evaluation

To train an accurate linear regression model, we need a way to quantify how good (or bad) our model performs. In machine learning, we call such performance-measuring functions *loss functions*. Several popular loss functions exist for regression problems.[i] To measure our model's performance, we'll use one of the most popular: mean-squared error (MSE).

Mean-Squared Error (MSE)

MSE quantifies how close a predicted value is to the true value, so we'll use it to quantify how close a regression line is to a set of points. MSE works by squaring the distance between each data point and the regression line (the red residuals in the graphs above), summing the squared values, and then dividing by the number of data points:

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

The name is quite literal: take the mean of the squared errors. The squaring of errors prevents negative and positive terms from canceling out in the sum,[i] and gives more weight to points further from the regression line, punishing outliers. In practice, we'll fit our regression model to a set training data, and evaluate it's performance using MSE on the test dataset.

R-Squared

Regression models may also be evaluated with the so-called *goodness of fit* measures, which summarize how well a model fits a set of data. The most popular goodness of fit measure for linear regression is r-squared, a metric that represents the percentage of the variance in y explained by our features x.[i] More specifically, r-squared measures the percentage of variance explained normalized against the baseline variance of our model (which is just the variance of the mean):

$$R^2 = 1 - \frac{\Sigma_{i=1}^{n}(y_i - \hat{y}_i)^2}{\Sigma_{i=1}^{n}(y_i - \bar{y})^2}$$

The highest possible value for r-squared is 1, representing a model that captures 100% of the variance. A negative r-squared means that our model is doing worse (capturing less variance) than a flat line through mean of our data would.

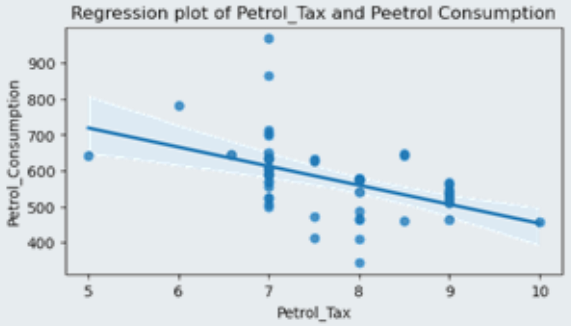
To build intuition for yourself, try changing the weight and bias terms below to see how the MSE and r-squared change across different model fits:

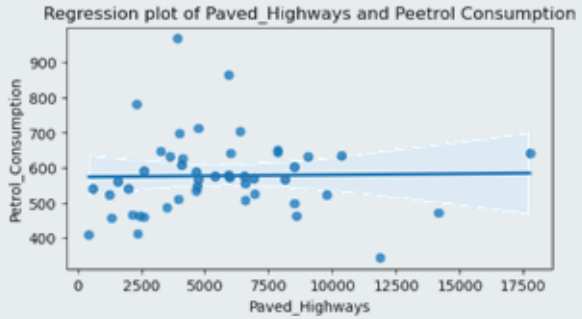
Multiple Linear Regression (MLR)

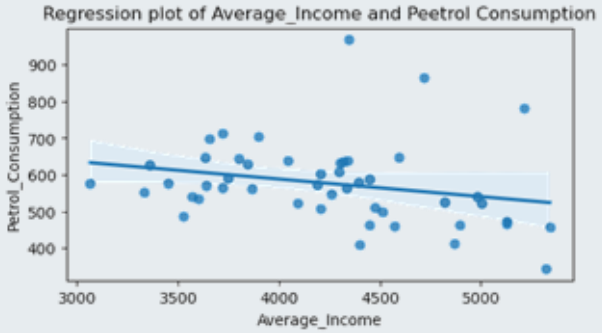
- **Multiple linear regression** is used to estimate the relationship between **two or more independent variables** and **one dependent variable**.

$$y = b_0 + b_1x_1 + b_2x_2 + \cdots + b_nx_n$$

- y: Dependent Variable
- b_0: Intercept
- b_1: First Independent Variable's Coefficient
- x_1: First Independent Variable



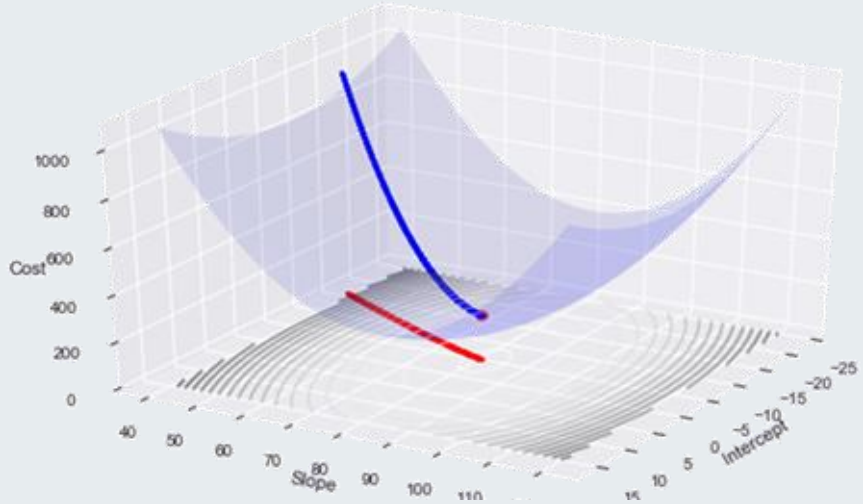


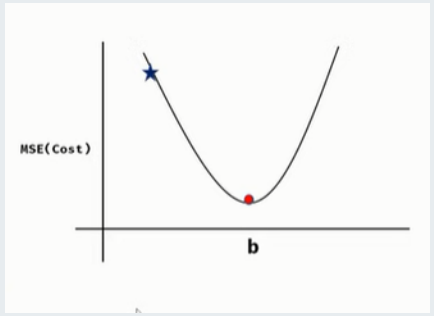


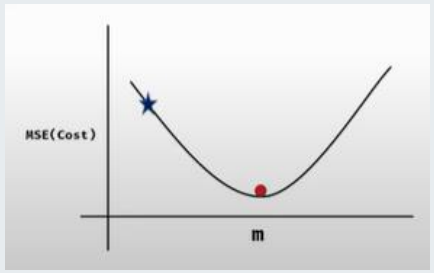
Reference –

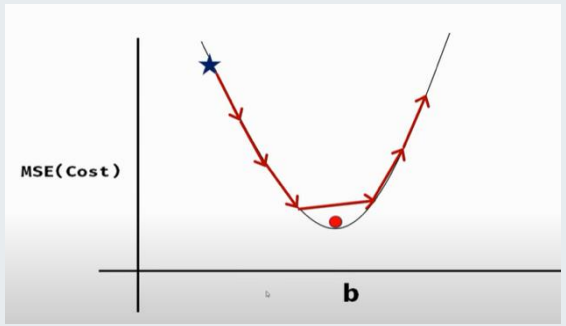
Cost Function

$$MSE\ or\ Cost = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - (mx_i + b)\right)^2$$

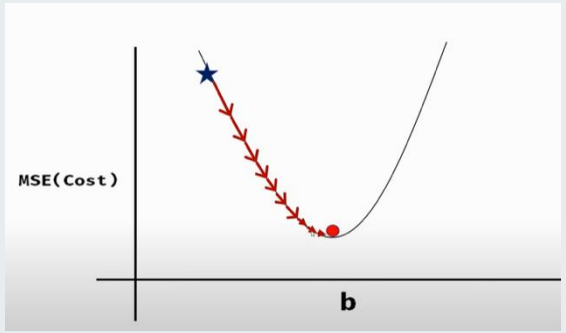








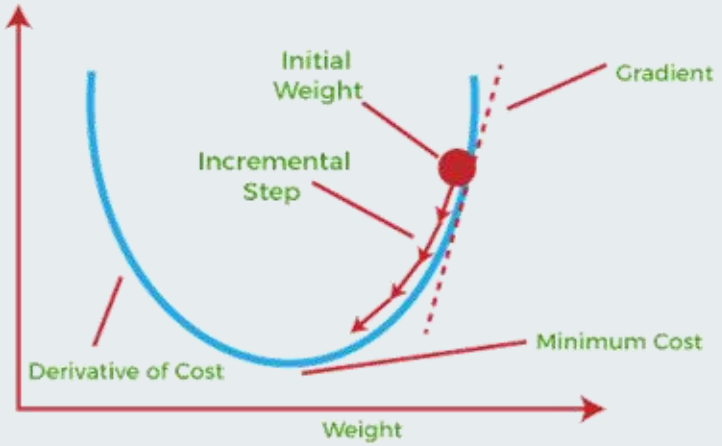
Constant step method

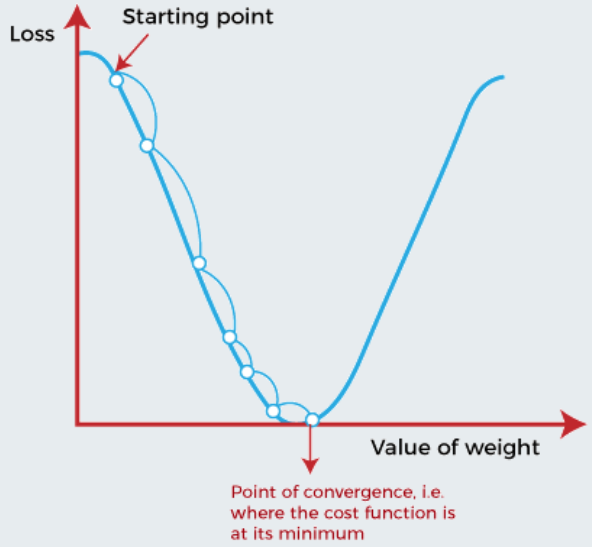


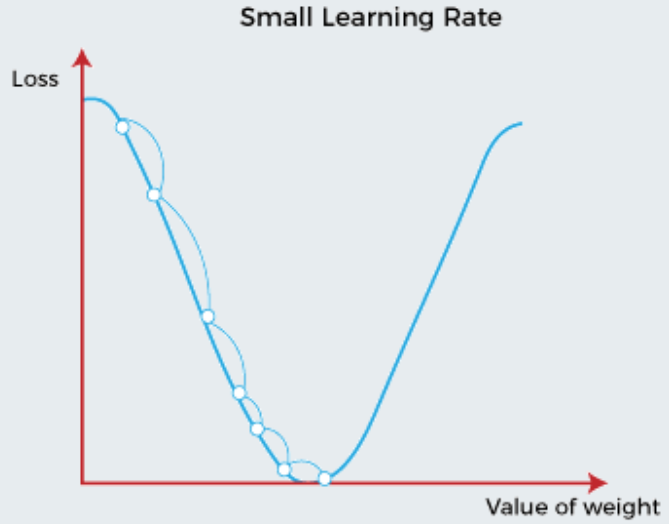
Gradient descent method

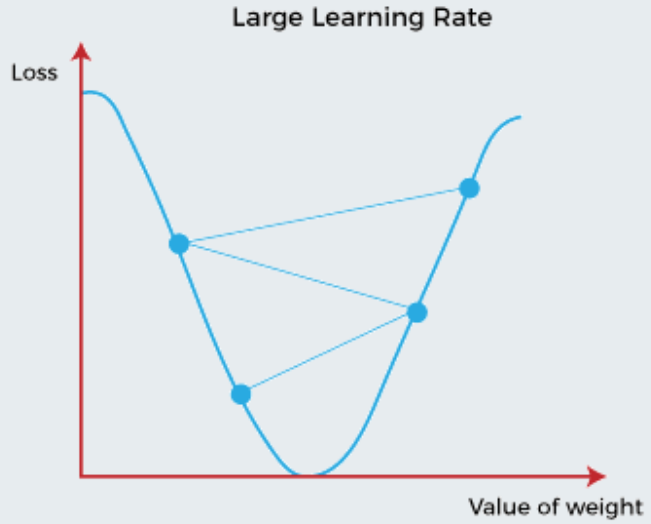
Reference – Google Images

Gradient Descent Method



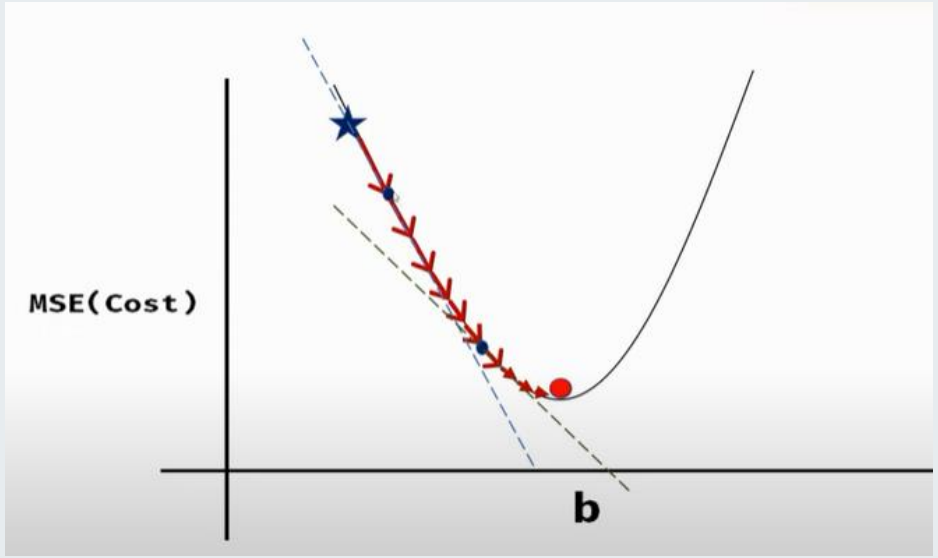






Reference –

Gradient Descent Method



$$MSE\ or\ Cost = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - (mx_i + b)\right)^2$$

$$\frac{\partial}{\partial m} = -\frac{2}{N}\sum_{i=1}^{N} x_i\left(y_i - (mx_i + b)\right)$$

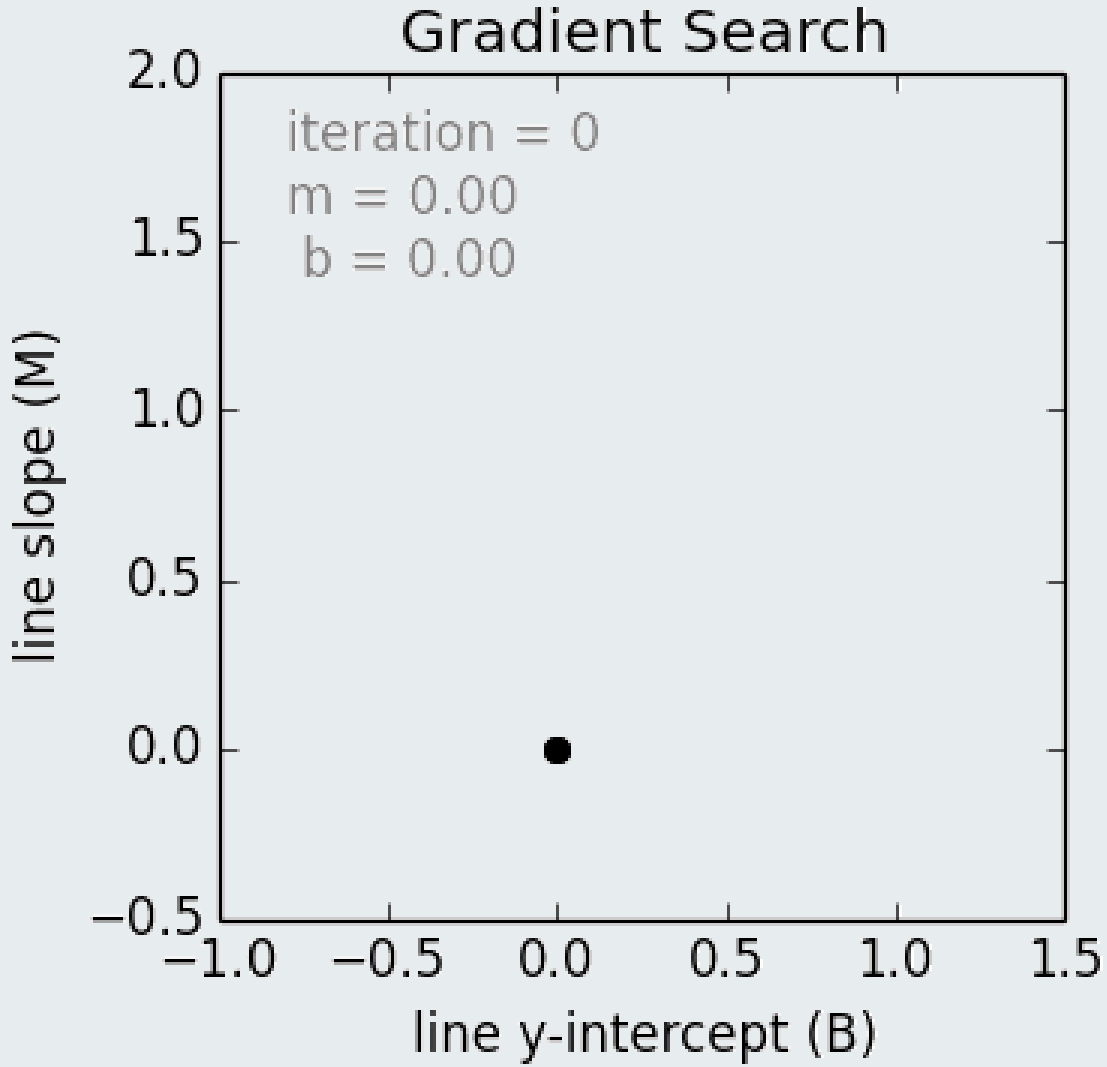
$$\frac{\partial}{\partial b} = -\frac{2}{N}\sum_{i=1}^{N}\left(y_i - (mx_i + b)\right)$$

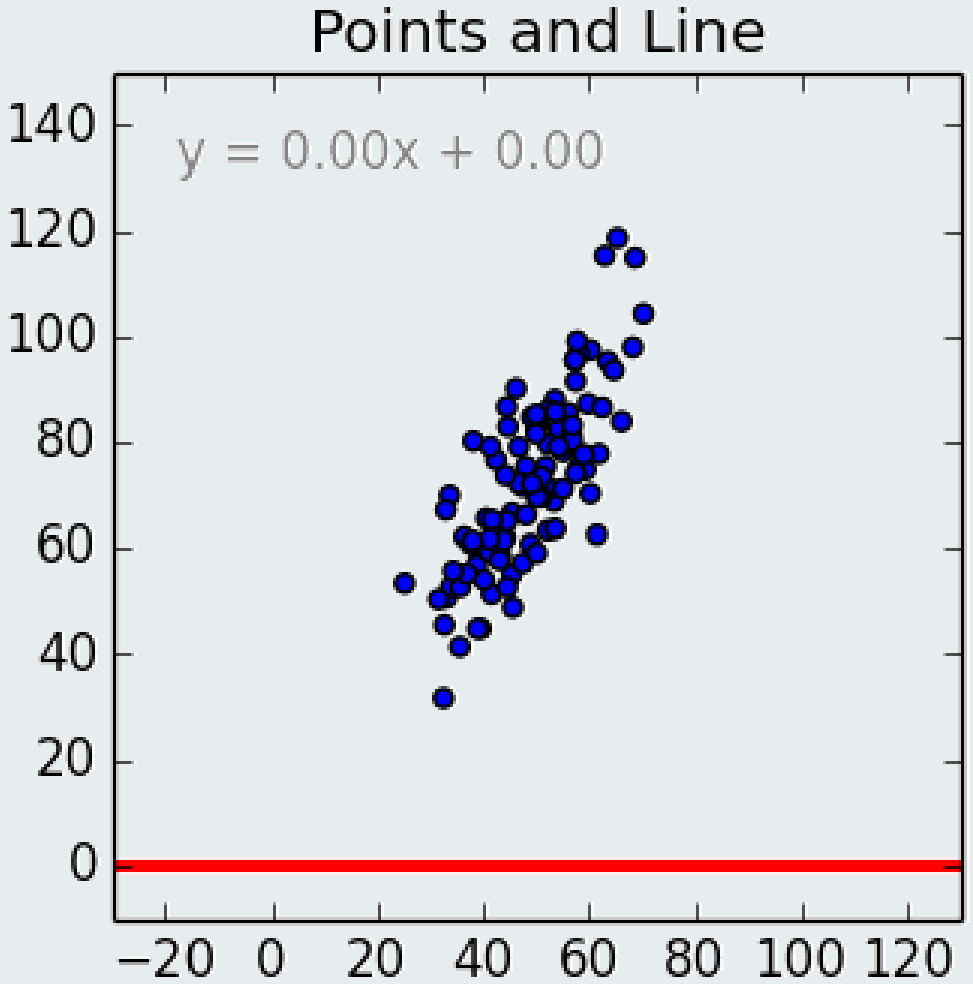
$$m = m - l * \frac{\partial}{\partial m}$$

$$b = b - l * \frac{\partial}{\partial b}$$

$l =$ Learning Rate

Reference –


Gradient Search
iteration = 0
m = 0.00
b = 0.00
line slope (M)
line y-intercept (B)
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−0.5
0.0
0.5
1.0
1.5


Points and Line
y = 0.00x + 0.00
140
120
100
80
60
40
20
0
−20
0
20
40
60
80
100
120

If you had studied longer, would your overall scores get any better?

- **Exploratory Data Analysis**

```
1 import pandas as pd
2 # Substitute the path_to_file content by the path to your student_scores.csv file
3 path_to_file = 'C:/Users/pc/OneDrive/Desktop/ML_Assignment/LinearRegression/student_scores.csv'
4 df = pd.read_csv(path_to_file)
5 df.head()
```

	Hours	Scores
0	2.5	21
1	5.1	47
2	3.2	27
3	8.5	75
4	3.5	30

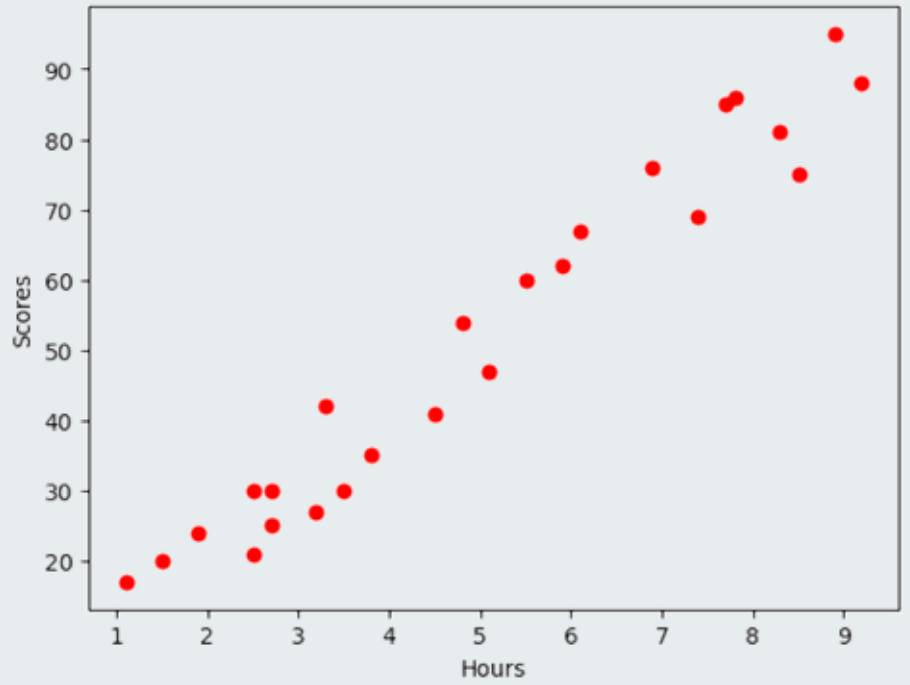
```
1 df.shape
```

```
(25, 2)
```

Reference –

- **Scatter Plot**

```
1  import matplotlib.pyplot as plt
2
3  %matplotlib inline
4  plt.xlabel("Hours")
5  plt.ylabel("Scores")
6  #plt.scatter(df.Hours,df.Scores, color="red", marker="+")
7  plt.scatter(df.Hours,df.Scores, color="red")
```



Reference –

Correlation between Variables

```
1 print(df.corr())
```

```
            Hours    Scores
Hours    1.000000  0.976191
Scores   0.976191  1.000000
```

In this table, Hours and Hours have a 1.0 (100%) correlation, just as Scores have a 100% correlation to Scores, naturally. Any variable will have a 1:1 mapping with itself! However, the correlation between Scores and Hours is 0.97. Anything above 0.8 is considered to be a strong positive correlation.

Description of Data

```
1 print(df.describe())
```

```
            Hours     Scores
count   25.000000  25.000000
mean     5.012000  51.480000
std      2.525094  25.286887
min      1.100000  17.000000
25%      2.700000  30.000000
50%      4.800000  47.000000
75%      7.400000  75.000000
max      9.200000  95.000000
```

Reference –

Linear Regression with Python's Scikit-learn

- **Data Processing**

```
y = df['Scores'].values.reshape(-1, 1)
X = df['Hours'].values.reshape(-1, 1)
```

Note: df['Column_Name'] returns a pandas Series. Some libraries can work on a Series just as they would on a NumPy array, but not all libraries have this awareness. In some cases, you'll want to extract the underlying NumPy array that describes your data. This is easily done via the values field of the Series.

```
print(df['Hours'].values) # [2.5 5.1 3.2 8.5 3.5 1.5 9.2 ... ]
print(df['Hours'].values.shape) # (25,)
```

```
[2.5 5.1 3.2 8.5 3.5 1.5 9.2 5.5 8.3 2.7 7.7 5.9 4.5 3.3 1.1 8.9 2.5 1.9
 6.1 7.4 2.7 4.8 3.8 6.9 7.8]
(25,)
```

It's expect a 2D input because the LinearRegression() class (more on it later) expects entries that may contain more than a single value (but can also be a single value). In either case - it has to be a 2D array, where each element (hour) is actually a 1-element array:

```
print(X.shape) # (25, 1)
print(X)       # [[2.5] [5.1]  [3.2] ... ]
```

```
(25, 1)
```

Reference –

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)
```

The method randomly takes samples respecting the percentage we've defined, but respects the X-y pairs, lest the sampling would totally mix up the relationship. Some common train-test splits are 80/20 and 70/30.

Since the sampling process is inherently random, we will always have different results when running the method. To be able to have the same results, or reproducible results, we can define a constant called SEED that has the value of 42.

```
SEED = 42
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = SEED)

```

Reference –

Training a Linear Regression Model

```
1 from sklearn.linear_model import LinearRegression
2 regressor = LinearRegression()
```

We have our train and test sets ready. Scikit-Learn has a plethora of model types we can easily import and train, LinearRegression being one of them

```
1 regressor.fit(X_train, y_train)
```

```
LinearRegression()
```

```
1 print(regressor.intercept_)
```

```
[2.82689235]
```

```
1 print(regressor.coef_)
```

```
[[9.68207815]]
```

Reference –

Let's check real quick whether this aligns with our guesstimation:

$hours=5$
$score=9.68207815*hours+2.82689235$
$score=51.2672831$

With 5 hours of study, you can expect around 51% as a score! Another way to interpret the intercept value is - if a student studies one hour more than they previously studied for an exam, they can expect to have an increase of 9.68% considering the score percentage that they had previously achieved.

Reference –

Making Predictions

To avoid running calculations ourselves, we could write our own formula that calculates the value:

```
def calc(slope, intercept, hours):
    return slope*hours+intercept

score = calc(regressor.coef_, regressor.intercept_, 9.5)
print(score) # [[94.80663482]]
```

```
[[94.80663482]]
```

However - a much handier way to predict new values using our model is to call on the predict() function:

```
# Passing 9.5 in double brackets to have a 2 dimensional array
score = regressor.predict([[9.5]])
print(score) # 94.80663482
```

```
[[94.80663482]]
```

To make predictions on the test data, we pass the X_test values to the predict() method. We can assign the results to the variable y_pred:

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

The y_pred variable now contains all the predicted values for the input values in the X_test. We can now compare the actual output values for X_test with the predicted values, by arranging them side by side in a dataframe structure:

```
df_preds = pd.DataFrame({'Hours':X_test.squeeze(),'Actual': y_test.squeeze(), 'Predicted': y_pred.squeeze()})
#squeeze helps arrays to convert in pandas dataframe object
print(df_preds)

```

```
   Hours  Actual  Predicted
0    8.3      81  83.188141
1    2.5      30  27.032088
2    2.5      21  27.032088
3    6.9      76  69.633232
4    5.9      62  59.951153
```

Evaluating the Model

- Mean Absolute Error (MAE)

$$mae = \frac{1}{n}\sum_{i=1}^{n} |Actual - Predicted|$$

- Mean Squared Error (MSE)

$$mse = \frac{1}{n}\sum_{i=1}^{n} (Actual - Predicted)^2$$

- Root Mean Squared Error (RMSE)

$$rmse = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (Actual - Predicted)^2}$$

Reference –

Luckily, we don't have to do any of the metrics calculations manually. The Scikit-Learn package already comes with functions that can be used to find out the values of these metrics for us. Let's find the values for these metrics using our test data. First, we will import the necessary modules for calculating the MAE and MSE errors. Respectively, the mean_absolute_error and mean_squared_error

```
1 from sklearn.metrics import mean_absolute_error, mean_squared_error
```

Now, we can calculate the MAE and MSE by passing the y_test (actual) and y_pred (predicted) to the methods. The RMSE can be calculated by taking the square root of the MSE, to to that, we will use NumPy's sqrt() method:

```
1 import numpy as np
```

```
1 mae = mean_absolute_error(y_test, y_pred)
2 mse = mean_squared_error(y_test, y_pred)
3 rmse = np.sqrt(mse)
```

```
print(f'Mean absolute error: {mae:.2f}')
print(f'Mean squared error: {mse:.2f}')
print(f'Root mean squared error: {rmse:.2f}')
```

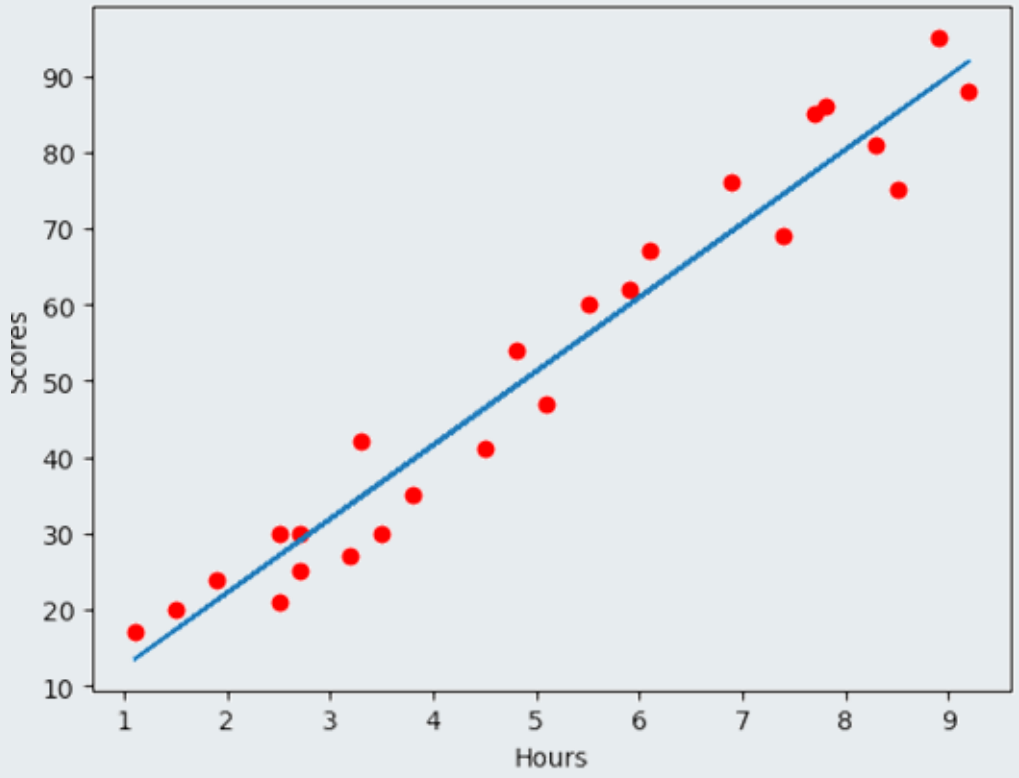
```
Mean absolute error: 3.92
Mean squared error: 18.94
Root mean squared error: 4.35
```

All of our errors are low - and we're missing the actual value by 4.35 at most (lower or higher), which is a pretty small range considering the data we have.

Reference –

How the final fit looks like ?

```
%matplotlib inline
plt.xlabel("Hours")
plt.ylabel("Scores")
#plt.scatter(df.Hours,df.Scores, color="red", marker="+")
plt.scatter(df.Hours,df.Scores, color="red")
plt.plot(df.Hours,regressor.predict(df['Hours'].values.reshape(-1,1)))
plt.show()
```



Reference –

Multiple Linear Regression (MLR)

- **Importing Libraries**

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 import numpy as np
4 from sklearn.model_selection import train_test_split
5 from sklearn.linear_model import LinearRegression
6 from sklearn.metrics import mean_absolute_error, mean_squared_error
```

- **Exploratory Data Analysis**

```
1 df= pd.read_csv('C:/Users/pc/OneDrive/Desktop/ML_Assignment/LinearRegression/MultipleLinearRegression/petrol_consumption.csv'
```

```
1 df.head()
```

	Petrol_Tax	Average_Income	Paved_Highways	Population_With_Driver_Licenses(%)	Petrol_Consumption
0	9.0	3571	1976	0.525	541
1	9.0	4092	1250	0.572	524
2	9.0	3865	1586	0.580	561
3	7.5	4870	2351	0.529	414
4	8.0	4399	431	0.544	410

```
1 df.shape
```

```
(48, 5)
```

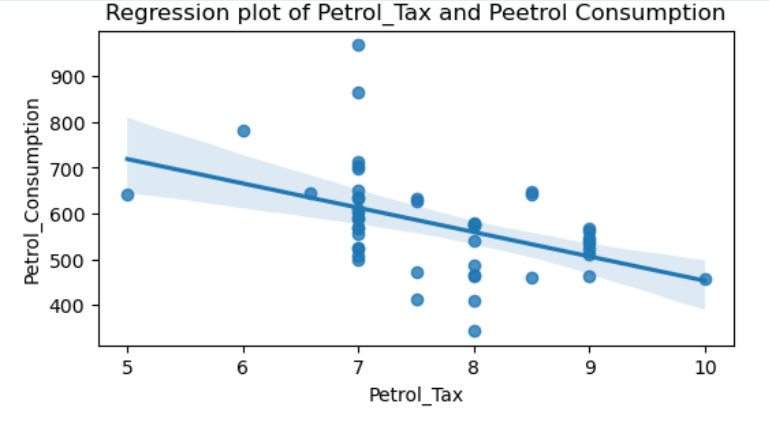
Reference –

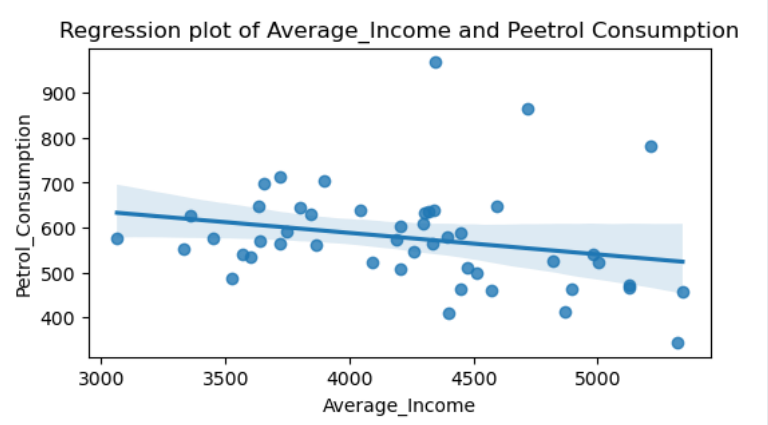
```
1 print(df.describe().round(2).T)
```

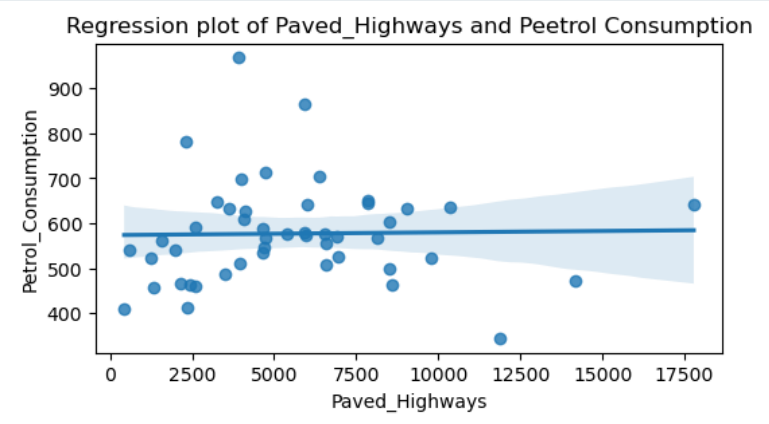
```
                                  count     mean      std      min      25%  \
Petrol_Tax                         48.0     7.67     0.95     5.00     7.00
Average_Income                     48.0  4241.83   573.62  3063.00  3739.00
Paved_Highways                     48.0  5565.42  3491.51   431.00  3110.25
Population_With_Driver_Licenses(%) 48.0     0.57     0.06     0.45     0.53
Petrol_Consumption                 48.0   576.77   111.89   344.00   509.50

                                    50%      75%       max
Petrol_Tax                         7.50     8.12     10.00
Average_Income                  4298.00  4578.75   5342.00
Paved_Highways                  4735.50  7156.00  17782.00
Population_With_Driver_Licenses(%)  0.56     0.60      0.72
Petrol_Consumption                568.50   632.75    968.00
```

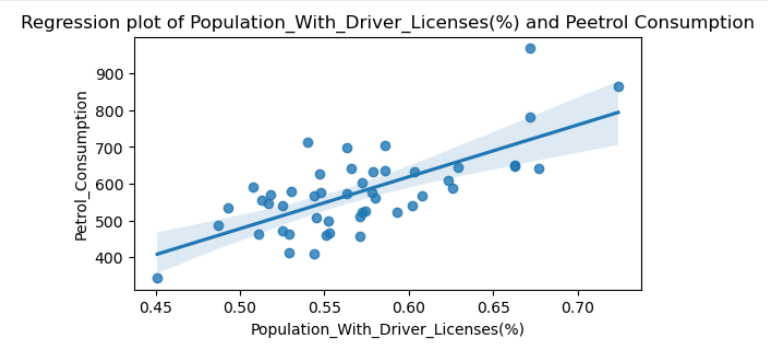
- **Visualization of data**

```
import seaborn as sns

variables=['Petrol_Tax','Average_Income','Paved_Highways','Population_With_Driver_Licenses(%)']
for var in variables:
    plt.figure(figsize=(6, 3)) # Creating a rectangle (figure) for each plot
    # Regression Plot also by default includes
    # best-fitting regression line
    # which can be turned off via `fit_reg=False`
    sns.regplot(x=var,y='Petrol_Consumption',data=df).set(title=f'Regression plot of {var} and Peetrol Consumption')
```

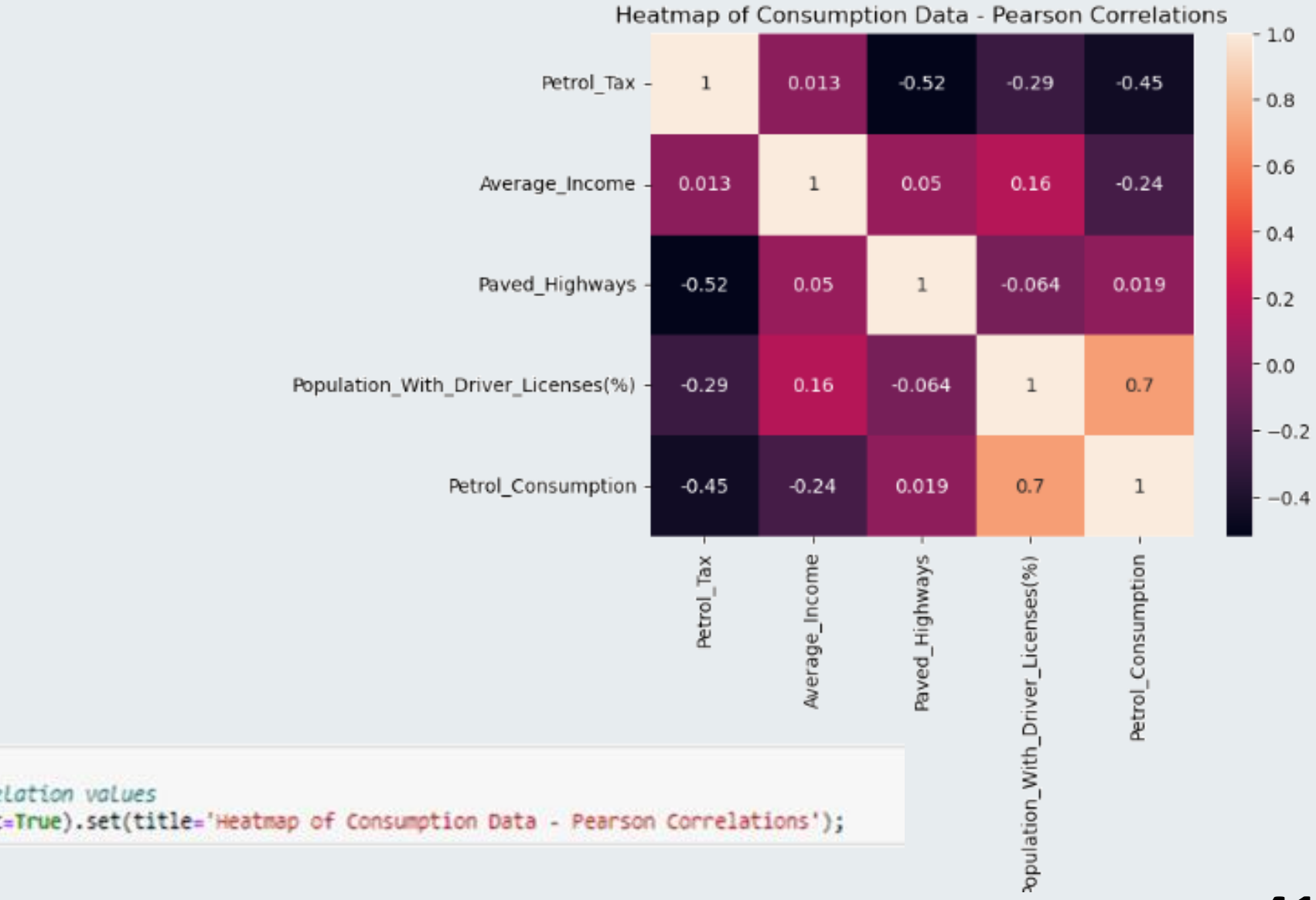








We can also calculate the correlation of the new variables, this time using Seaborn's heatmap() to help us spot the strongest and weaker correlations based on warmer (reds) and cooler (blues) tones:




```
correlations = df.corr()
# annot=True displays the correlation values
sns.heatmap(correlations, annot=True).set(title='Heatmap of Consumption Data - Pearson Correlations');
```

Reference –

- **Preparing the data**

```
y = df['Petrol_Consumption']
X = df[['Average_Income', 'Paved_Highways',
        'Population_With_Driver_Licenses(%)', 'Petrol_Tax']]
```

After setting our X and y sets, we can divide our data into train and test sets. We will be using the same seed and 20% of our data for training:

```
SEED=42
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.2,
                                                    random_state=SEED)
```

- **Training the Multivariate Model**

```
regressor = LinearRegression()
regressor.fit(X_train, y_train)
```

```
LinearRegression()
```

```
print(regressor.intercept_)
print(regressor.coef_)
```

```
361.4508790666836
[-5.65355145e-02 -4.38217137e-03  1.34686930e+03 -3.69937459e+01]
```

Reference –

```
1 feature_names = X.columns
```

```
1 print(feature_names)
```

```
Index(['Average_Income', 'Paved_Highways',
       'Population_With_Driver_Licenses(%)', 'Petrol_Tax'],
      dtype='object')
```

```
1 feature_names = X.columns
2 model_coefficients = regressor.coef_
3 coefficients_df = pd.DataFrame(data = model_coefficients,
4                                index = feature_names,
5                                columns = ['Coefficient value'])
6 print(coefficients_df)
```

	Coefficient value
Average_Income	-0.056536
Paved_Highways	-0.004382
Population_With_Driver_Licenses(%)	1346.869298
Petrol_Tax	-36.993746

Reference –

- **Making Predictions with the Multivariate Regression Model**

```
1 y_pred = regressor.predict(X_test)
```

Now, that we have our test predictions, we can better compare them with the actual output values for X_test by organizing them in a DataFrameformat:

```
1 results = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred})
2 print(results)
```

```
    Actual   Predicted
27     631  606.692665
40     587  673.779442
26     577  584.991490
43     591  563.536910
24     460  519.058672
37     704  643.461003
12     525  572.897614
19     640  687.077036
4      410  547.609366
25     566  530.037630
```

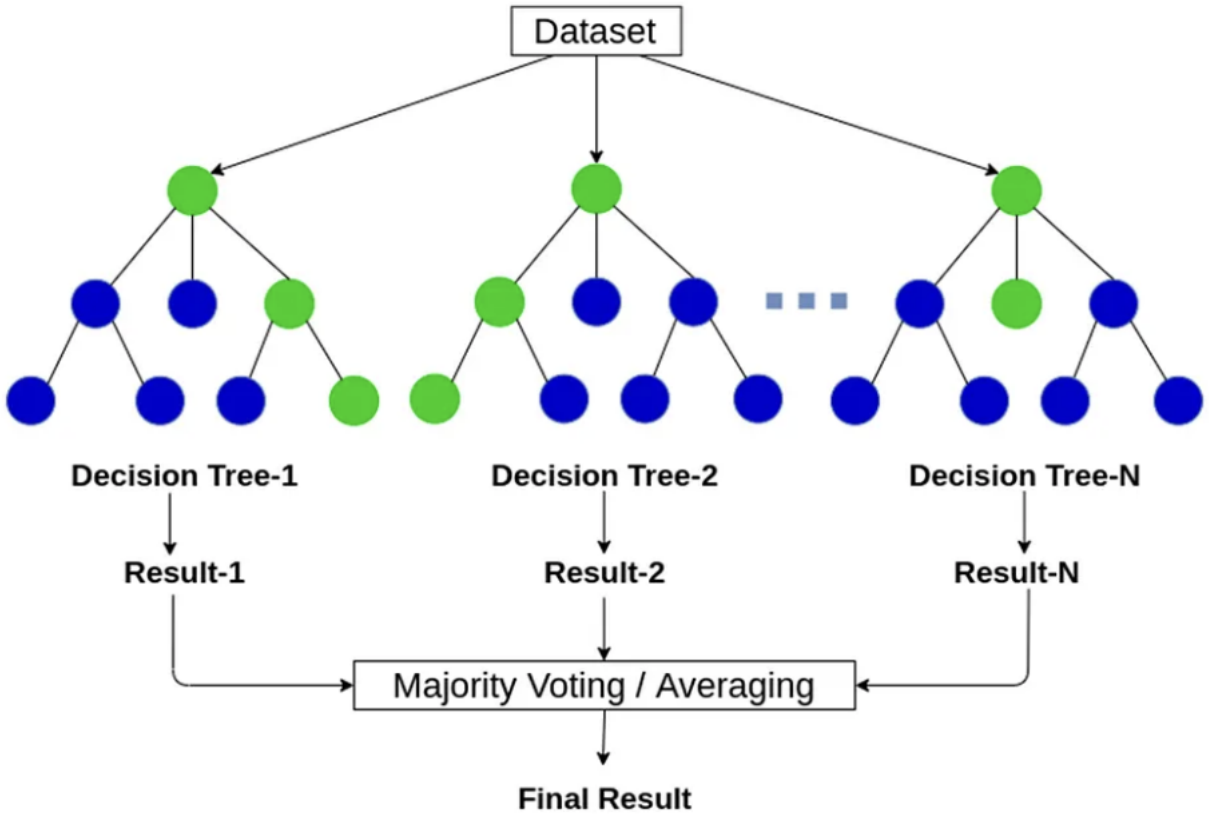
Reference –

```python
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)

print(f'Mean absolute error: {mae:.2f}')
print(f'Mean squared error: {mse:.2f}')
print(f'Root mean squared error: {rmse:.2f}')
```

```
Mean absolute error: 53.47
Mean squared error: 4083.26
Root mean squared error: 63.90
```

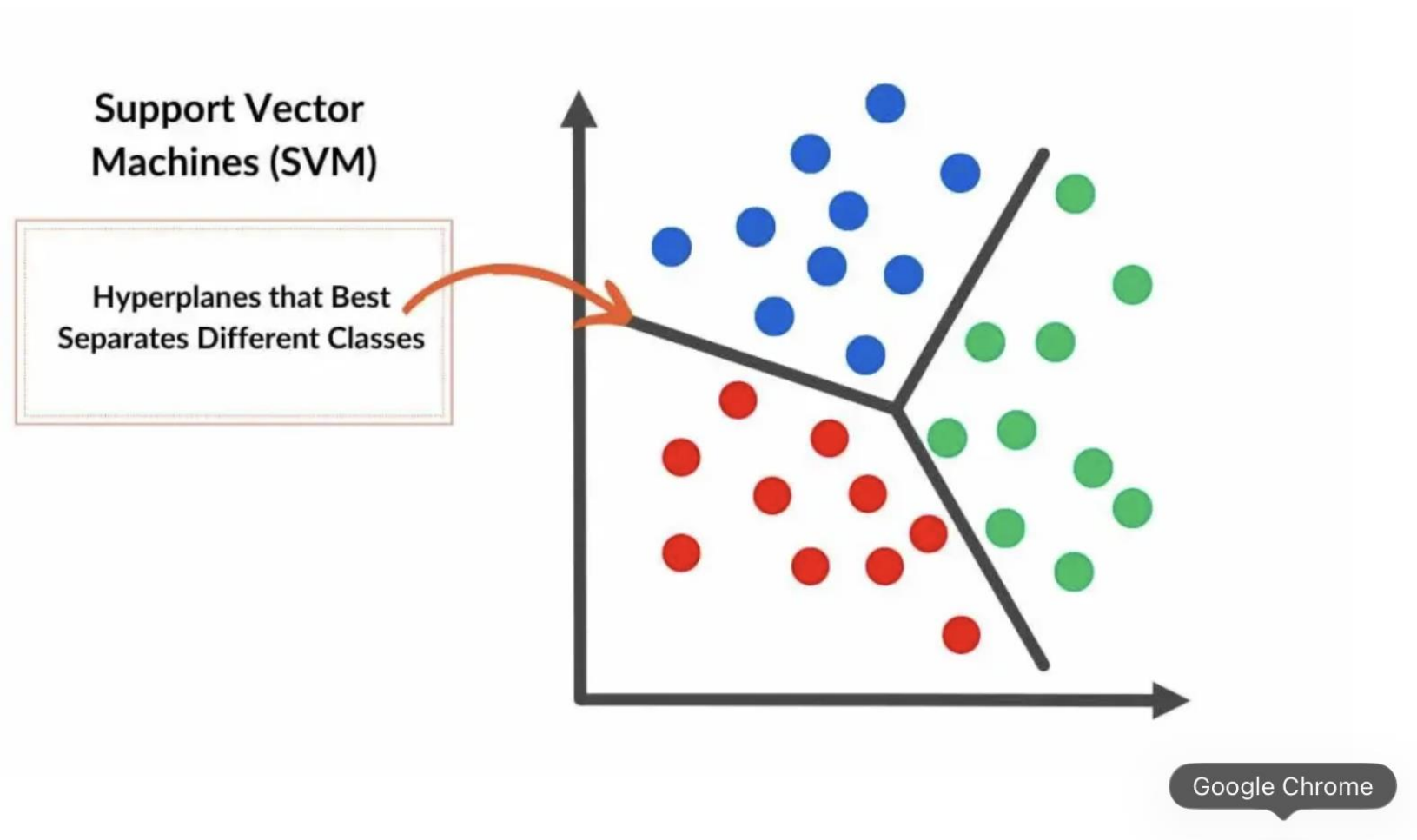
Reference –

Random Forest

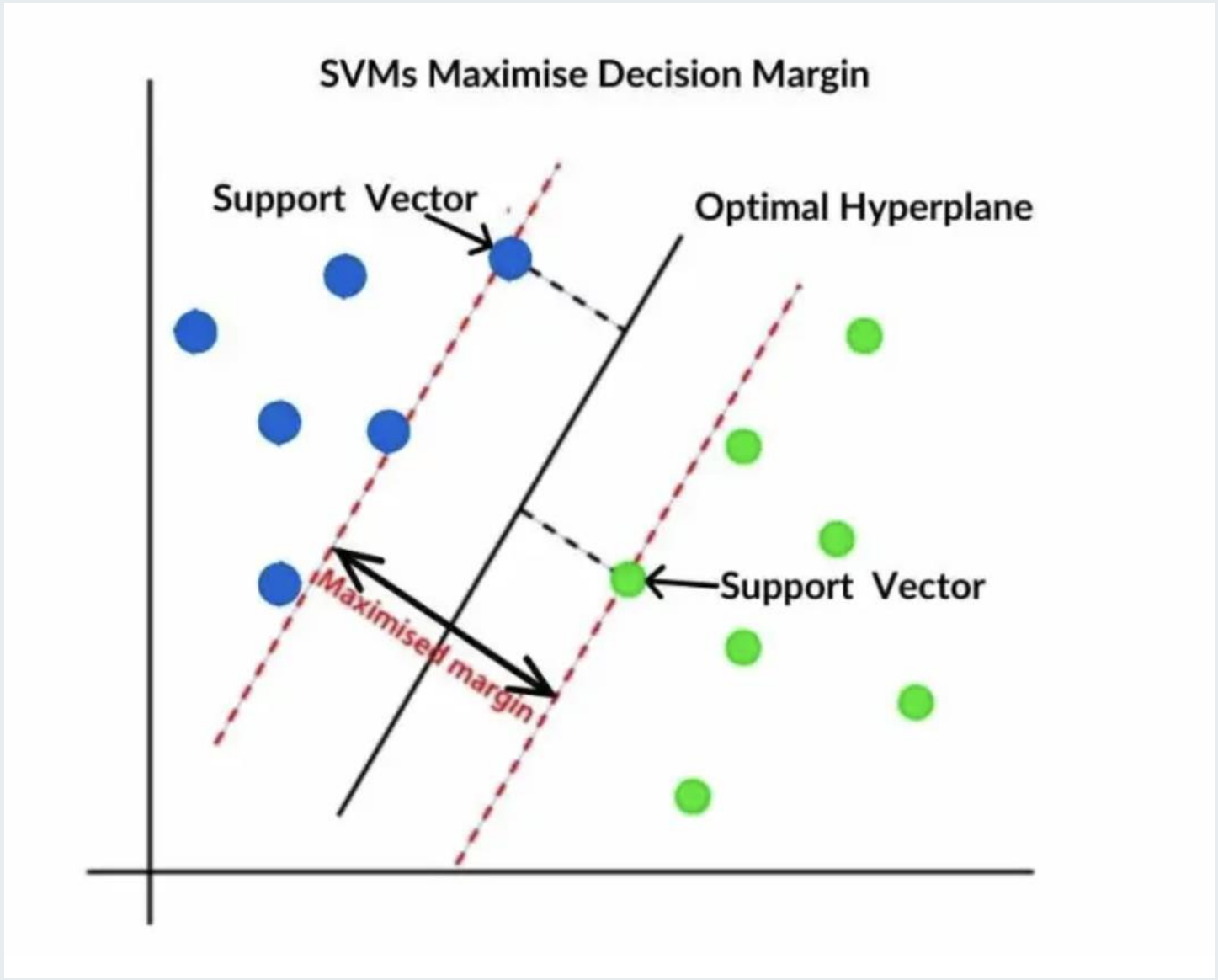


Random forest — workflow

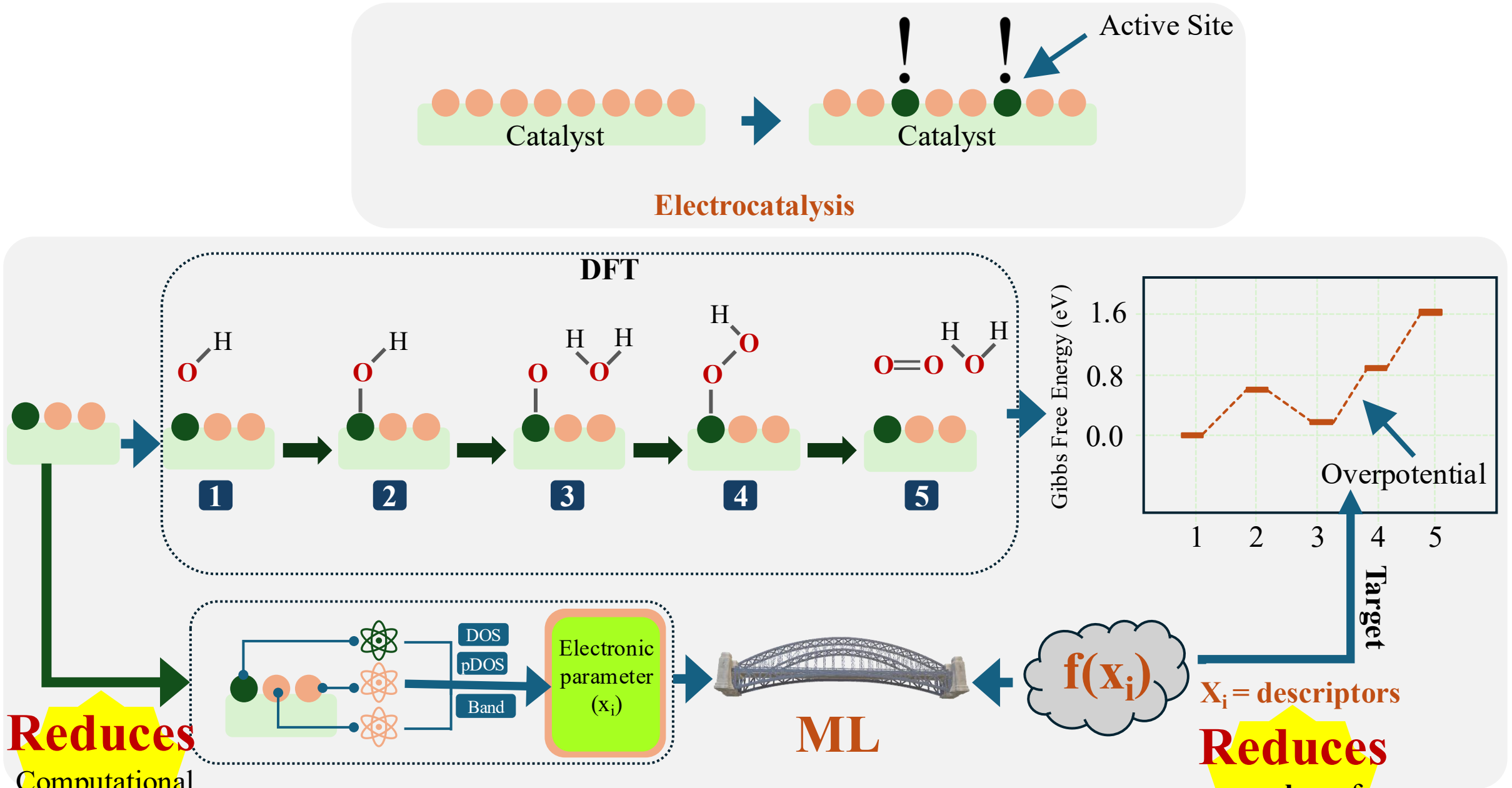
https://medium.com/@abhishekjainindore24/everything-about-random-forest-90c106d63989



https://spotintelligence.com/2024/05/08/support-vector-regression-svr/https://spotintelligence.com/2024/05/08/support-vector-regression-svr/



https://spotintelligence.com/2024/05/08/support-vector-regression-svr/https://spotintelligence.com/2024/05/08/support-vector-regression-svr/


Active Site
Catalyst
Catalyst
Electrocatalysis
DFT
1
2
3
4
5
Gibbs Free Energy (eV)
1.6
0.8
0.0
1
2
3
4
5
Overpotential
Target
DOS
pDOS
Band
Electronic parameter (x_i)
ML
$f(x_i)$
X_i = descriptors
Reduces
Computational
cost and time
Reduces
number of
Calculation
Makes LIFE easier

Reference –

THANK YOU