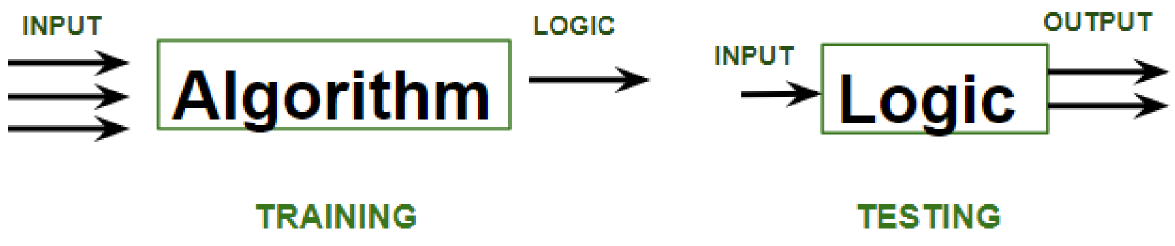
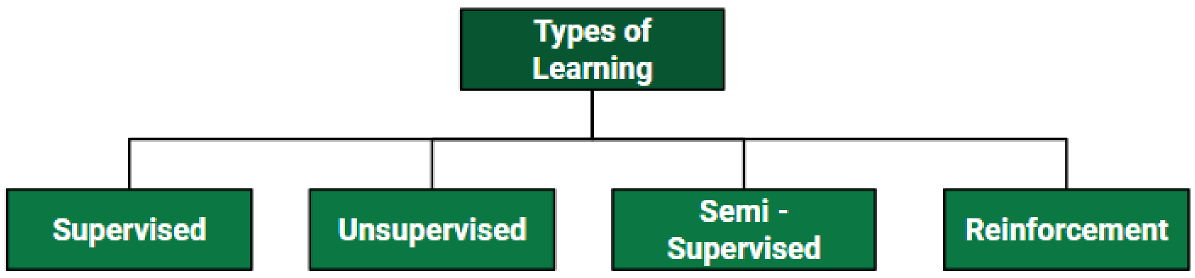
Approaches of ML and Python Basics

Types of Learning – Supervised Learning

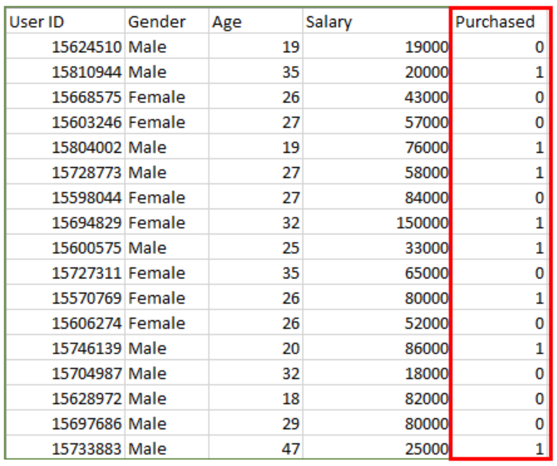
Let us discuss what is learning for a machine is as shown below media as follows:



A machine is said to be learning from **past Experiences**(data feed-in) with respect to some class of **tasks** if its **Performance** in a given Task improves with the Experience. For example, assume that a machine has to predict whether a customer will buy a specific product let's say "Antivirus" this year or not. The machine will do it by looking at the **previous knowledge/past experiences** i.e the data of products that the customer had bought every year and if he buys Antivirus every year, then there is a high probability that the customer is going to buy an antivirus this year as well. This is how machine learning works at the basic conceptual level.

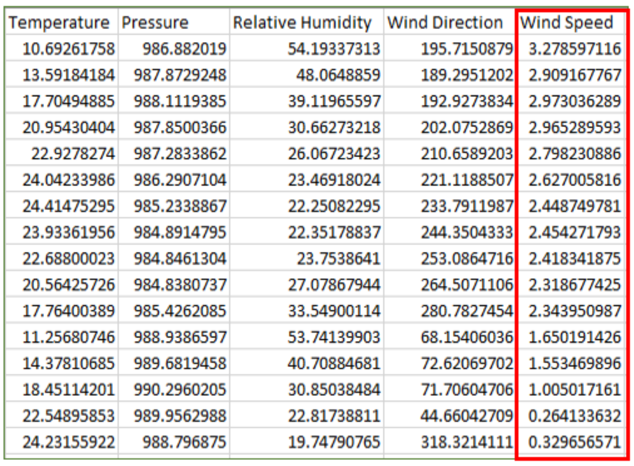


Supervised learning is when the model is getting trained on a labelled dataset. A **labelled** dataset is one that has both input and output parameters. In this type of learning both training and validation, datasets are labelled as shown in the figures below.



User ID	Gender	Age	Salary	Purchased
15624510	Male	19	19000	0
15810944	Male	35	20000	1
15668575	Female	26	43000	0
15603246	Female	27	57000	0
15804002	Male	19	76000	1
15728773	Male	27	58000	1
15598044	Female	27	84000	0
15694829	Female	32	150000	1
15600575	Male	25	33000	1
15727311	Female	35	65000	0
15570769	Female	26	80000	1
15606274	Female	26	52000	0
15746139	Male	20	86000	1
15704987	Male	32	18000	0
15628972	Male	18	82000	0
15697686	Male	29	80000	0
15733883	Male	47	25000	1

Figure A: CLASSIFICATION



Temperature	Pressure	Relative Humidity	Wind Direction	Wind Speed
10.69261758	986.882019	54.19337313	195.7150879	3.278597116
13.59184184	987.8729248	48.0648859	189.2951202	2.909167767
17.70494885	988.1119385	39.11965597	192.9273834	2.973036289
20.95430404	987.8500366	30.66273218	202.0752869	2.965289593
22.9278274	987.2833862	26.06723423	210.6589203	2.798230886
24.04233986	986.2907104	23.46918024	221.1188507	2.627005816
24.41475295	985.2338867	22.25082295	233.7911987	2.448749781
23.93361956	984.8914795	22.35178837	244.3504333	2.454271793
22.68800023	984.8461304	23.7538641	253.0864716	2.418341875
20.56425726	984.8380737	27.07867944	264.5071106	2.318677425
17.76400389	985.4262085	33.54900114	280.7827454	2.343950987
11.25680746	988.9386597	53.74139903	68.15406036	1.650191426
14.37810685	989.6819458	40.70884681	72.62069702	1.553469896
18.45114201	990.2960205	30.85038484	71.70604706	1.005017161
22.54895853	989.9562988	22.81738811	44.66042709	0.264133632
24.23155922	988.796875	19.74790765	318.3214111	0.329656571

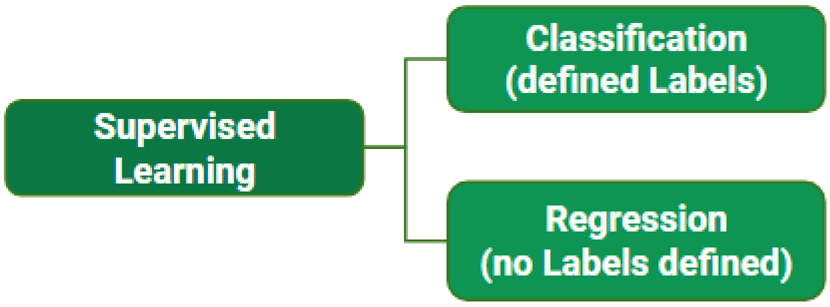
Figure B: REGRESSION

Both the above figures have labelled data set as follows:

- **Figure A:** It is a dataset of a shopping store that is useful in predicting whether a customer will purchase a particular product under consideration or not based on his/ her gender, age, and salary. **Input:** Gender, Age, Salary **Output:** Purchased i.e. 0 or 1; 1 means yes the customer will purchase and 0 means that the customer won't purchase it.
- **Figure B:** It is a Meteorological dataset that serves the purpose of predicting wind speed based on different parameters. **Input:** Dew Point, Temperature, Pressure, Relative Humidity, Wind Direction **Output:** Wind Speed

Training the system: While training the model, data is usually split in the ratio of 80:20 i.e. 80% as training data and the rest as testing data. In training data, we feed input as well as output for 80% of data. The model learns from training data only. We use different machine learning algorithms(which we will discuss in detail in the next articles) to build our model. Learning means that the model will build some logic of its own.

Once the model is ready then it is good to be tested. At the time of testing, the input is fed from the remaining 20% of data that the model has never seen before, the model will predict some value and we will compare it with the actual output and calculate the accuracy.


Supervised Learning
Classification (defined Labels)
Regression (no Labels defined)

Types of Supervised Learning:

Supervised learning is a machine learning technique that is widely used in various fields such as finance, healthcare, marketing, and more. It is a form of machine learning in which the algorithm is trained on labeled data to make predictions or decisions based on the data inputs.

In supervised learning, the algorithm learns a mapping between the input and output data. This mapping is learned from a labeled dataset, which consists of pairs of input and output data. The algorithm tries to learn the relationship between the input and output data so that it can make accurate predictions on new, unseen data.

The labeled dataset used in supervised learning consists of input features and corresponding output labels. The input features are the attributes or characteristics of the data that are used to make predictions, while the output labels are the desired outcomes or targets that the algorithm tries to predict.

Supervised learning is typically divided into two main categories: regression and classification. In regression, the algorithm learns to predict a continuous output value, such as the price of a house or the temperature of a city. In classification, the algorithm learns to predict a categorical output variable or class label, such as whether a customer is likely to purchase a product or not.

One of the primary advantages of supervised learning is that it allows for the creation of complex models that can make accurate predictions on new data. However, supervised learning requires large amounts of labeled training data to be effective. Additionally, the quality and representativeness of the training data can have a significant impact on the accuracy of the model.

Supervised learning can be further classified into two categories:

Regression: In regression, the target variable is a continuous value. The goal of regression is to predict the value of the target variable based on the input variables. Linear regression, polynomial regression, and decision trees are some of the examples of regression algorithms.

Classification: In classification, the target variable is a categorical value. The goal of classification is to predict the class or category of the target variable based on the input variables. Some examples of classification algorithms include logistic regression, decision trees, support vector machines, and neural networks.

Supervised learning can be further divided into several different types, each with its own unique characteristics and applications. Here are some of the most common types of supervised learning:**(Algorithms used in Supervised Learning)**

Linear Regression

Linear regression is a type of regression algorithm that is used to predict a continuous output value. It is one of the simplest and most widely used algorithms in supervised learning. In linear regression, the algorithm tries to find a linear relationship between the input features and the output value. The output value is predicted based on the weighted sum of the input features.

Logistic Regression

Logistic regression is a type of classification algorithm that is used to predict a binary output variable. It is commonly used in machine learning applications where the output variable is either true or false, such as in fraud detection or spam filtering. In logistic regression, the algorithm tries to find a linear relationship between the input features and the output variable. The output variable is then transformed using a logistic function to produce a probability value between 0 and 1.

Decision Trees

A decision tree is a type of algorithm that is used for both classification and regression tasks. It is a tree-like structure that is used to model decisions and their possible consequences. Each internal node in the tree represents a decision, while each leaf node represents a possible outcome. Decision trees can be used to model complex relationships between input features and output variables.

Random Forests

Random forests are an ensemble learning technique that is used for both classification and regression tasks. They are made up of multiple decision trees that work together to make predictions. Each tree in the forest is trained on a different subset of the input features and data. The final prediction is made by aggregating the predictions of all the trees in the forest

- hine (SVM)
- Random Forest

Getting started with Classification

As the name suggests, Classification is the task of "classifying things" into sub-categories. But, by a machine! If that doesn't sound like much, imagine your computer being able to differentiate between you and a stranger. Between a potato and a tomato. Between an A grade and an F. Now, it sounds interesting now. Classification is part of supervised machine learning in which we put labeled data for training. In Machine Learning and Statistics, Classification is the problem of identifying to which of a set of categories (subpopulations), a new observation belongs, on the basis of a training set of data containing observations and whose categories membership is known.

What is Classification?

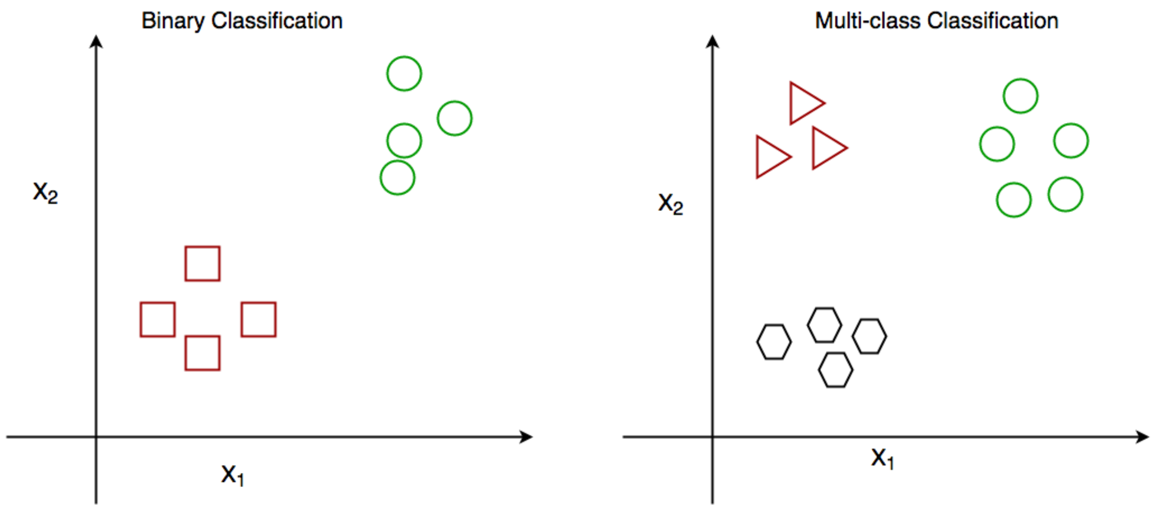
Classification is a process of categorizing data or objects into predefined classes or categories based on their features or attributes. In **machine learning**, classification is a type of **supervised learning** technique where an algorithm is trained on a labeled dataset to predict the class or category of new, unseen data.

The main objective of classification is to build a model that can accurately assign a label or category to a new observation based on its features. For example, a classification model might be trained on a dataset of images labeled as either dogs or cats and then used to predict the class of new, unseen images of dogs or cats based on their features such as color, texture, and shape.

Types of Classification

Classification is of two types:

1. **Binary Classification**: In binary classification, the goal is to classify the input into one of two classes or categories. Example – On the basis of the given health conditions of a person, we have to determine whether the person has a certain disease or not.
2. **Multiclass Classification**: In multi-class classification, the goal is to classify the input into one of several classes or categories. For Example – On the basis of data about different species of flowers, we have to determine which specie our observation belongs to.



Binary vs Multi class classification

Types of classification algorithms

There are various types of classifiers. Some of them are :

- **Linear Classifiers:** Linear models create a linear decision boundary between classes. They are simple and computationally efficient. Some of the linear **classification** models are as follows:
    - **Logistic Regression**
    - **Support Vector Machines having kernel = 'linear'**
    - **Single-layer Perceptron**
    - Stochastic Gradient Descent (SGD) Classifier
- **Non-linear Classifiers:** Non-linear models create a non-linear decision boundary between classes. They can capture more complex relationships

between the input features and the target variable. Some of the non-linear **classification** models are as follows:

- **K-Nearest Neighbours**
- **Kernel SVM**
- **Naive Bayes**
- **Decision Tree Classification**
- **Ensemble learning classifiers:**
    - **Random Forests,**
    - **AdaBoost,**
    - **Bagging Classifier,**
    - **Voting Classifier,**
    - **ExtraTrees Classifier**
- **Multi-layer Artificial Neural Networks**

Type of Learners in Classifications Algorithm

In machine learning, classification learners can also be classified as either “lazy” or “eager” learners.

- **Lazy Learners**: Lazy Learners are also known as instance-based learners, lazy learners do not learn a model during the training phase. Instead, they simply store the training data and use it to classify new instances at prediction time. It is very fast at prediction time because it does not require computations during the predictions. it is less effective in high-dimensional spaces or when the number of training instances is large. Examples of lazy learners include k-nearest neighbors and case-based reasoning.
- **Eager Learners**: Eager Learners are also known as model-based learners, eager learners learn a model from the training data during the training phase and use this model to classify new instances at prediction time. It is more effective in high-dimensional spaces having large training datasets. Examples of eager learners include decision trees, random forests, and support vector machines.

Classification model Evaluations

Evaluating a classification model is an important step in machine learning, as it helps to assess the performance and generalization ability of the model on new, unseen data. There are several metrics and techniques that can be used to evaluate a classification model, depending on the specific problem and requirements. Here are some commonly used evaluation metrics:

- **Classification Accuracy:** The proportion of correctly classified instances over the total number of instances in the test set. It is a simple and intuitive metric but can be misleading in imbalanced datasets where the majority class dominates the accuracy score.
- **Confusion matrix:** A table that shows the number of true positives, true negatives, false positives, and false negatives for each class, which can be used to calculate various evaluation metrics.
- **Precision and Recall:** Precision measures the proportion of true positives over the total number of predicted positives, while recall measures the proportion of true positives over the total number of actual positives. These metrics are useful in scenarios where one class is more important than the other, or when there is a trade-off between false positives and false negatives.
- **F1-Score:** The harmonic mean of precision and recall, calculated as 2 x (precision x recall) / (precision + recall). It is a useful metric for imbalanced datasets where both precision and recall are important.
- **ROC curve and AUC:** The Receiver Operating Characteristic (ROC) curve is a plot of the true positive rate (recall) against the false positive rate (1-specificity) for different threshold values of the classifier's decision function. The Area Under the Curve (AUC) measures the overall performance of the classifier, with values ranging from 0.5 (random guessing) to 1 (perfect classification).
- **Cross-validation:** A technique that divides the data into multiple folds and trains the model on each fold while testing on the others, to obtain a more robust estimate of the model's performance.

It is important to choose the appropriate evaluation metric(s) based on the specific problem and requirements, and to avoid overfitting by evaluating the model on independent test data.

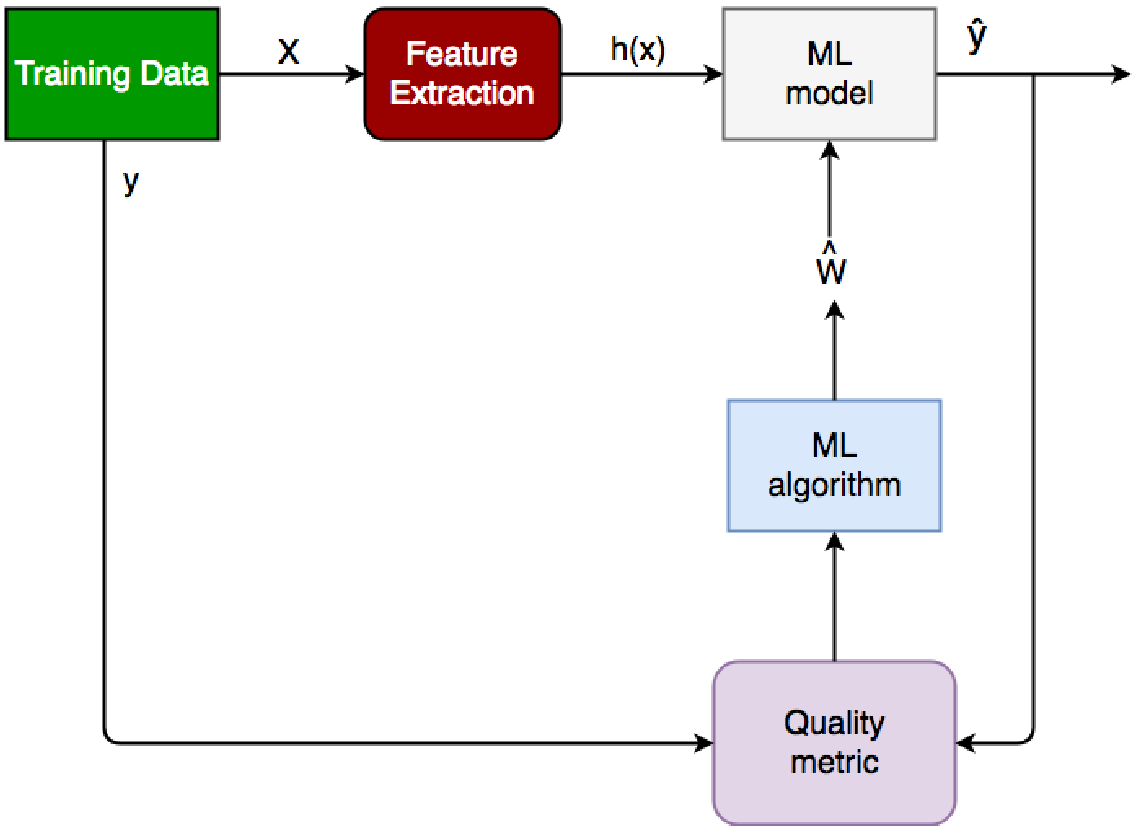
How does classification work?

The basic idea behind classification is to train a model on a labeled dataset, where the input data is associated with their corresponding output labels, to learn the patterns and relationships between the input data and output labels. Once the model is trained, it can be used to predict the output labels for new unseen data.

The classification process typically involves the following steps:

1. **Understanding the problem:** Before getting started with classification, it is important to understand the problem you are trying to solve. What are the class labels you are trying to predict? What is the relationship between the input data and the class labels?
    - Suppose we have to predict whether a patient has a certain disease or not, on the basis of 7 independent variables, called features. This means, there can be only two possible outcomes:
        - The patient has the disease, which means "**True**".
        - The patient has no disease. which means "**False**".
    - This is a binary classification problem.
2. **Data preparation:** Once you have a good understanding of the problem, the next step is to prepare your data. This includes collecting and preprocessing the data and splitting it into training, validation, and test sets. In this step, the data is cleaned, preprocessed, and transformed into a format that can be used by the classification algorithm.
    - **X:** It is the independent feature, in the form of an N*M matrix. N is the no. of observations and M is the number of features.
    - **y:** An N vector corresponding to predicted classes for each of the N observations.
3. **Feature Extraction:** The relevant features or attributes are extracted from the data that can be used to differentiate between the different classes.
    - Suppose our input X has 7 independent features, having only 5 features influencing the label or target values remaining 2 are negligibly or not correlated, then we will use only these 5 features only for the model training.

4. **Model Selection:** There are many different models that can be used for classification, including **logistic regression, decision trees, support vector machines (SVM), or neural networks**. It is important to select a model that is appropriate for your problem, taking into account the size and complexity of your data, and the computational resources you have available.
5. **Model Training:** Once you have selected a model, the next step is to train it on your training data. This involves adjusting the parameters of the model to minimize the error between the predicted class labels and the actual class labels for the training data.
6. **Model Evaluation:** Evaluating the model: After training the model, it is important to evaluate its performance on a validation set. This will give you a good idea of how well the model is likely to perform on new, unseen data.
    - Log Loss or Cross-Entropy Loss, Confusion Matrix, Precision, Recall, and AUC-ROC curve are the quality metrics used for measuring the performance of the model.
7. **Fine-tuning the model:** If the model's performance is not satisfactory, you can fine-tune it by adjusting the parameters, or trying a different model.
8. **Deploying the model:** Finally, once we are satisfied with the performance of the model, we can deploy it to make predictions on new data. it can be used for real world problem.



Classification Lifecycle

Applications of Classification Algorithm

Classification algorithms are widely used in many real-world applications across various domains, including:

- Email spam filtering
- Credit risk assessment
- Medical diagnosis
- Image classification
- Sentiment analysis.
- Fraud detection
- Quality control
- Recommendation systems

Implementation:

Let’s get a hands-on experience with how Classification works.
We are going to study various Classifiers
and see a rather simple analytical comparison of their performance on a well-known, standard data set, the Iris data set.

Requirements for running the given script:

- **Python**
- **Scipy and Numpy**
- **Pandas**
- **Scikit-learn**

Example

```
# Importing the required libraries
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn import datasets
from sklearn import svm
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB

# import the iris dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target

# splitting X and y into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=1)

# GAUSSIAN NAIVE BAYES
gnb = GaussianNB()
# train the model
gnb.fit(X_train, y_train)
# make predictions
gnb_pred = gnb.predict(X_test)
```

```
# print the accuracy
print("Accuracy of Gaussian Naive Bayes: ",
    accuracy_score(y_test, gnb_pred))

# DECISION TREE CLASSIFIER
dt = DecisionTreeClassifier(random_state=0)
# train the model
dt.fit(X_train, y_train)
# make predictions
dt_pred = dt.predict(X_test)
# print the accuracy
print("Accuracy of Decision Tree Classifier: ",
    accuracy_score(y_test, dt_pred))

# SUPPORT VECTOR MACHINE
svm_clf = svm.SVC(kernel='linear') # Linear Kernel
# train the model
svm_clf.fit(X_train, y_train)
# make predictions
svm_clf_pred = svm_clf.predict(X_test)
# print the accuracy
print("Accuracy of Support Vector Machine: ",
    accuracy_score(y_test, svm_clf_pred))
```

Output:

```
Accuracy of Gaussian Naive Bayes:  0.9333333333333333
Accuracy of Decision Tree Classifier:  0.9555555555555556
Accuracy of Support Vector Machine:  1.0
```

Conclusion:

Classification is a very vast field of study. Even though it comprises a small part of Machine Learning as a whole, it is one of the most important ones.

That’s all for now. In the next article, we will see how Classification works in practice and get our hands dirty with Python Code.

Types of Regression Techniques in ML

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 linear regression. It tries to fit data with the best hyperplane which goes through the points.

What is Regression Analysis?

Regression Analysis is a statistical process for estimating the relationships between the dependent variables or criterion variables and one or more independent variables or predictors. Regression analysis is generally used when we deal with a dataset that has the target variable in the form of continuous data. Regression analysis explains the changes in criteria in relation to changes in select predictors. The conditional expectation of the criteria is based on predictors where the average value of the dependent variables is given when the independent variables are changed. Three major uses for regression analysis are determining the strength of predictors, forecasting an effect, and trend forecasting.

What is the purpose to use Regression Analysis?

There are times when we would like to analyze the effect of different independent features on the target or what we say dependent features. This helps us make decisions that can affect the target variable in the desired direction. Regression analysis is heavily based on **statistics** and hence gives quite reliable results due to this reason only regression models are used to find the linear as well as non-linear relation between the independent and the dependent or target variables.

Types of Regression Techniques

Along with the development of the machine learning domain regression analysis techniques have gained popularity as well as developed manifold from just y = mx + c. There are several types of regression techniques, each suited for different types of data and different types of relationships. The main types of regression techniques are:

1. **Linear Regression**
2. **Polynomial Regression**
3. **Stepwise Regression**
4. **Decision Tree Regression**
5. **Random Forest Regression**
6. **Support Vector Regression**
7. **Ridge Regression**
8. **Lasso Regression**
9. **ElasticNet Regression**
10. **Bayesian Linear Regression**

Linear Regression

Linear regression is used for predictive analysis. **Linear regression** is a linear approach for modeling the relationship between the criterion or the scalar response and the multiple predictors or explanatory variables. Linear regression focuses on the conditional probability distribution of the response given the values of the predictors. For linear regression, there is a danger of **overfitting**. The formula for linear regression is:

> Syntax:y = θx + bwhere,θ – It is the model weights or parametersb – It is known as the bias.

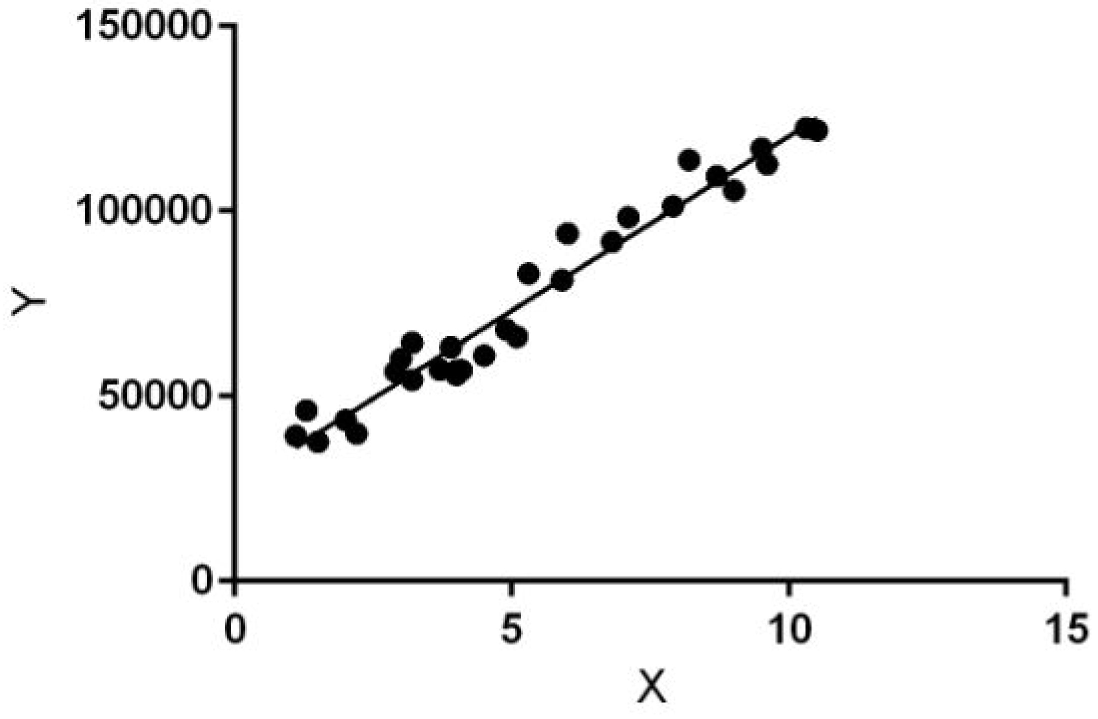
This is the most basic form of regression analysis and is used to model a linear relationship between a single dependent variable and one or more independent variables.

Linear regression is a type of supervised machine learning algorithm that computes the linear relationship between a dependent variable and one or more independent features. When the number of the independent feature, is 1 then it is known as Univariate Linear regression, and in the case of more than one feature, it is known as multivariate linear regression. The goal of the algorithm is to find the best linear equation that can predict the value of the dependent variable based on the independent variables. The equation provides a straight line that represents the relationship between the dependent and independent variables. The slope of the line indicates how much the dependent variable changes for a unit change in the independent variable(s).

Linear regression is used in many different fields, including finance, economics, and psychology, to understand and predict the behavior of a particular variable. For example, in finance, linear regression might be used to understand the relationship between a company's stock price and its earnings or to predict the future value of a currency based on its past performance.

One of the most important supervised learning tasks is regression. In regression set of records are present with X and Y values and these values are used to learn a function so if you want to predict Y from an unknown X this learned function can be used. In regression we have to find the value of Y, So, a function is required that predicts continuous Y in the case of regression given X as independent features.

Here Y is called a dependent or target variable and X is called an independent variable also known as the predictor of Y. There are many types of functions or modules that can be used for regression. A linear function is the simplest type of function. Here, X may be a single feature or multiple features representing the problem.



Linear Regression

Linear regression performs the task to predict a dependent variable value (y) based on a given independent variable (x)). 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.

Assumption for Linear Regression Model

Linear regression is a powerful tool for understanding and predicting the behavior of a variable, however, it needs to meet a few conditions in order to be accurate and dependable solutions.

1. **Linearity**: The independent and dependent variables have a linear relationship with one another. This implies that changes in the dependent variable follow those in the independent variable(s) in a linear fashion.
2. **Independence**: The observations in the dataset are independent of each other. This means that the value of the dependent variable for one observation does not depend on the value of the dependent variable for another observation.
3. **Homoscedasticity**: Across all levels of the independent variable(s), the variance of the errors is constant. This indicates that the amount of the independent variable(s) has no impact on the variance of the errors.
4. **Normality**: The errors in the model are normally distributed.
5. **No multicollinearity**: There is no high correlation between the independent variables. This indicates that there is little or no correlation between the independent variables.

Hypothesis function for Linear Regression :

As we have assumed earlier that our independent feature is the experience i.e X and the respective salary Y is the dependent variable. Let's assume there is a linear relationship between X and Y then the salary can be predicted using:

$$\hat{Y} = \theta_1 + \theta_2 X$$
$$\text{OR}$$
$$\hat{y}_i = \theta_1 + \theta_2 x_i$$

Here,

- are labels to data (Supervised learning)

$$y_i \epsilon Y \ \ (i = 1, 2, \cdots, n)$$

- are the input independent training data (univariate – one input variable(parameter))

$$x_i \epsilon X \ \ (i = 1, 2, \cdots, n)$$

- are the predicted values.

$$\hat{y_i} \epsilon \hat{Y} \ \ (i = 1, 2, \cdots, n)$$

The model gets the best regression fit line by finding the best θ1 and θ2 values.

- **θ1:** intercept
- **θ2:** coefficient of x

Once we find the best θ1 and θ2 values, we get the best-fit line. So when we are finally using our model for prediction, it will predict the value of y for the input value of x.

Cost function

The cost function or the loss function is nothing but the error or difference between the predicted value

$$\hat{Y}$$

and the true value Y. It is the

Mean Squared Error (MSE)

between the predicted value and the true value. The cost function (J) can be written as:

$$\text{Cost function}(J) = \frac{1}{n} \sum_{n}^{i} (\hat{y_i} - y_i)^2$$

How to update θ1 and θ2 values to get the best-fit line?

To achieve the best-fit regression line, the model aims to predict the target value

$$\hat{Y}$$

such that the error difference between the predicted value

$$\hat{Y}$$

and the true value Y is minimum. So, it is very important to update the θ

1

and θ

2

values, to reach the best value that minimizes the error between the predicted y value (pred) and the true y value (y).

$$minimize\frac{1}{n}\sum_{i=1}^{n}(\hat{y_i}-y_i)^2$$

<u>**Gradient Descent**</u>**:**

A linear regression model can be trained using the optimization algorithm gradient descent by iteratively modifying the model's parameters to reduce the mean squared error (MSE) of the model on a training dataset. To update θ1 and θ2 values in order to reduce the Cost function (minimizing RMSE value) and achieve the best-fit line the model uses Gradient Descent. The idea is to start with random θ1 and θ2values and then iteratively update the values, reaching minimum cost.

A gradient is nothing but a derivative that defines the effects on outputs of the function with a little bit of variation in inputs.

Let's differentiate the cost function(J) with respect to

$$\theta_1$$

$$
\begin{aligned}
J'_{\theta_1} &= \frac{\partial J(\theta_1, \theta_2)}{\partial \theta_1} \\
&= \frac{\partial}{\partial \theta_1}\left[\frac{1}{n}\left(\sum_{i=1}^{n}(\hat{y}_i - y_i)^2\right)\right] \\
&= \frac{1}{n}\left[\sum_{i=1}^{n} 2(\hat{y}_i - y_i)\left(\frac{\partial}{\partial \theta_1}(\hat{y}_i - y_i)\right)\right] \\
&= \frac{1}{n}\left[\sum_{i=1}^{n} 2(\hat{y}_i - y_i)\left(\frac{\partial}{\partial \theta_1}(\theta_1 + \theta_2 x_i - y_i)\right)\right] \\
&= \frac{1}{n}\left[\sum_{i=1}^{n} 2(\hat{y}_i - y_i)\left(1 + 0 - 0\right)\right] \\
&= \frac{1}{n}\left[\sum_{i=1}^{n}(\hat{y}_i - y_i)\left(2\right)\right] \\
&= \frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)
\end{aligned}
$$

Let's differentiate the cost function(J) with respect to

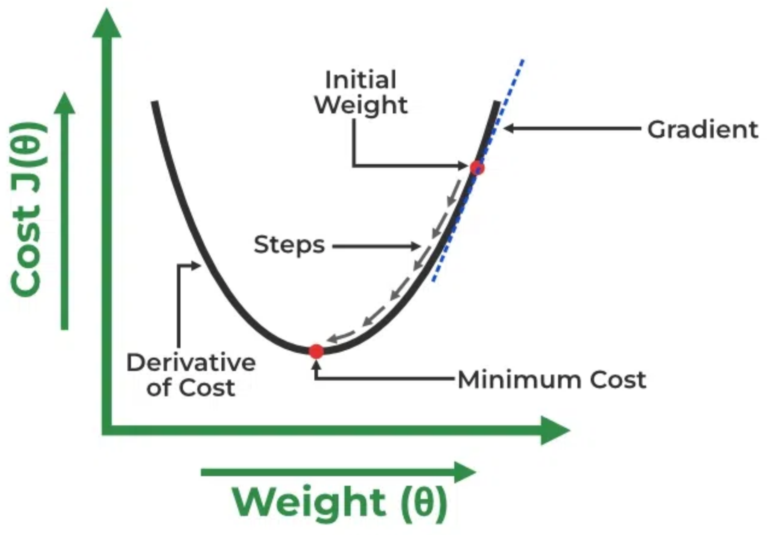
$$\theta_2$$

$$
\begin{aligned}
J'_{\theta_2} &= \frac{\partial J(\theta_1,\theta_2)}{\partial \theta_2} \\
&= \frac{\partial}{\partial \theta_2}\left[\frac{1}{n}\left(\sum_{i=1}^{n}(\hat{y}_i - y_i)^2\right)\right] \\
&= \frac{1}{n}\left[\sum_{i=1}^{n}2(\hat{y}_i - y_i)\left(\frac{\partial}{\partial \theta_2}(\hat{y}_i - y_i)\right)\right] \\
&= \frac{1}{n}\left[\sum_{i=1}^{n}2(\hat{y}_i - y_i)\left(\frac{\partial}{\partial \theta_2}(\theta_1 + \theta_2 x_i - y_i)\right)\right] \\
&= \frac{1}{n}\left[\sum_{i=1}^{n}2(\hat{y}_i - y_i)\left(0 + x_i - 0\right)\right] \\
&= \frac{1}{n}\left[\sum_{i=1}^{n}(\hat{y}_i - y_i)\left(2x_i\right)\right] \\
&= \frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)\cdot x_i
\end{aligned}
$$

Finding the coefficients of a linear equation that best fits the training data is the objective of linear regression. By moving in the direction of the Mean Squared Error negative gradient with respect to the coefficients, the coefficients can be changed. And the respective intercept and coefficient of X will be if

$$\alpha$$

is the learning rate.



Gradient Descent

$$\theta_1 = \theta_1 - \alpha\left(J'_{\theta_1}\right)$$
$$= \theta_1 - \alpha\left(\frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)\right)$$
$$\theta_2 = \theta_2 - \alpha\left(J'_{\theta_2}\right)$$
$$= \theta_2 - \alpha\left(\frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)\cdot x_i\right)$$

Build the Linear Regression model from Scratch

Import the necessary libraries:

```
import pandas as pd
```

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
import matplotlib.axes as ax
```

Load the dataset and separate input and Target variables

Dataset Link: [https://github.com/AshishJangra27/Machine-Learning-with-Python-GFG/tree/main/Linear%20Regression]

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.axes as ax

data = pd.read_csv('data_for_lr.csv')

# Drop the missing values
data = data.dropna()

# training dataset and labels
train_input = np.array(data.x[0:500]).reshape(500,1)
train_output = np.array(data.y[0:500]).reshape(500,1)

# valid dataset and labels
test_input = np.array(data.x[500:700]).reshape(199,1)
test_output = np.array(data.y[500:700]).reshape(199,1)
```

Build the Linear Regression Model

Steps:

- In forward propagation, Linear regression function Y=mx+x is applied by initially assigning random value of parameter (m & c).
- The we have written the function to finding the cost function i.e the mean

```
class LinearRegression:
    def __init__(self):
        self.parameters = {}

    def forward_propagation(self, train_input):
        m = self.parameters['m']
        c = self.parameters['c']
        predictions = np.multiply(m, train_input) + c
        return predictions
    def cost_function(self, predictions, train_output):
        cost = np.mean((train_output - predictions) ** 2)
        return cost

    def backward_propagation(self, train_input, train_outpu
t, predictions):
        derivatives = {}
        df = (train_output - predictions) * -1
        dm = np.mean(np.multiply(train_input, df))
        dc = np.mean(df)
        derivatives['dm'] = dm
        derivatives['dc'] = dc
        return derivatives

    def update_parameters(self, derivatives, learning_rat
e):
        self.parameters['m'] = self.parameters['m'] - learn
ing_rate * derivatives['dm']
        self.parameters['c'] = self.parameters['c'] - learn
ing_rate * derivatives['dc']

    def train(self, train_input, train_output, learning_rat
e, iters):
        #initialize random parameters
        self.parameters['m'] = np.random.uniform(0,1) * -1
        self.parameters['c'] = np.random.uniform(0,1) * -1

        #initialize loss
        self.loss = []

        #iterate
        for i in range(iters):
            #forward propagation
            predictions = self.forward_propagation(train_in
put)

            #cost function
            cost = self.cost_function(predictions, train_ou
tput)

            #append loss and print
            self.loss.append(cost)
            print("Iteration = {}, Loss = {}".format(i+1, c
ost))

            #back propagation
            derivatives = self.backward_propagation(train_i
nput, train_output, predictions)

            #update parameters
            self.update_parameters(derivatives, learning_ra
te)

        return self.parameters, self.loss
```

Trained the model

```
#Example usage
linear_reg = LinearRegression()
parameters, loss = linear_reg.train(train_input, train_outp
ut, 0.0001, 20)
```

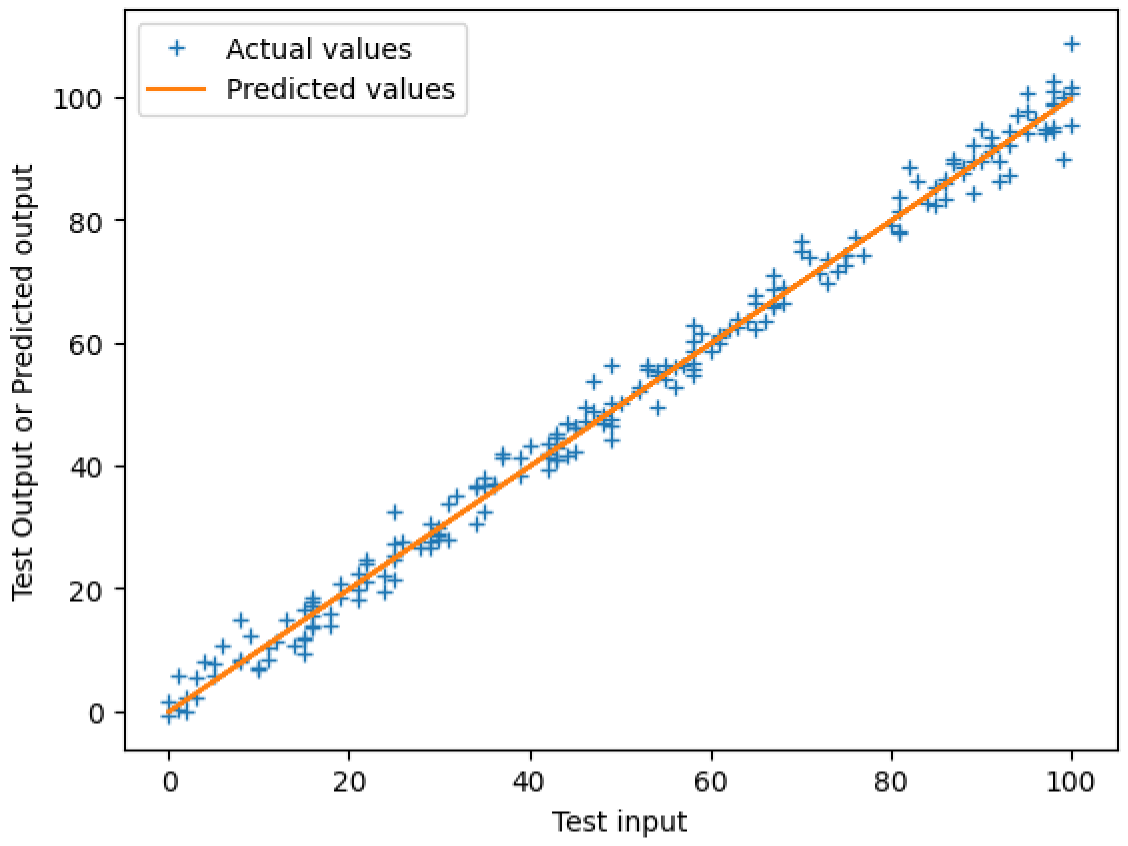
Output:

```
Iteration = 1, Loss = 5363.981028641572
Iteration = 2, Loss = 2437.9165904342512
Iteration = 3, Loss = 1110.3579137897523
Iteration = 4, Loss = 508.043071737168
Iteration = 5, Loss = 234.7721607488976
Iteration = 6, Loss = 110.78884574712548
Iteration = 7, Loss = 54.53747840152165
Iteration = 8, Loss = 29.016170730218153
Iteration = 9, Loss = 17.43712517102535
Iteration = 10, Loss = 12.183699375121314
Iteration = 11, Loss = 9.800214272338595
Iteration = 12, Loss = 8.718824440889573
Iteration = 13, Loss = 8.228196676299069
Iteration = 14, Loss = 8.005598315794709
Iteration = 15, Loss = 7.904605192804647
Iteration = 16, Loss = 7.858784500769819
Iteration = 17, Loss = 7.837995601770647
Iteration = 18, Loss = 7.828563654998014
Iteration = 19, Loss = 7.824284370030002
Iteration = 20, Loss = 7.822342853430061
```

Final Prediction and Plot the regression line

```
#Prediction on test data
y_pred = test_input*parameters['m'] + parameters['c']

# Plot the regression line with actual data pointa
plt.plot(test_input, test_output, '+', label='Actual value
s')
plt.plot(test_input, y_pred, label='Predicted values')
plt.xlabel('Test input')
plt.ylabel('Test Output or Predicted output')
plt.legend()
plt.show()
```

Output:



Best fit Linear regression line with actual values

Polynomial Regression

This is an extension of linear regression and is used to model a non-linear relationship between the dependent variable and independent variables. Here as well syntax remains the same but now in the input variables we include some polynomial or higher degree terms of some already existing features as well. Linear regression was only able to fit a linear model to the data at hand but with **polynomial features**, we can easily fit some non-linear relationship between the target as well as input features.

Polynomial Regression is a form of linear regression in which the relationship between the independent variable x and dependent variable y is modeled as an *nth-degree* polynomial. Polynomial regression fits a nonlinear relationship between the value of x and the corresponding conditional mean of y, denoted E(y | x).

What is a Polynomial Regression?

- There are some relationships that a researcher will hypothesize is curvilinear. Clearly, such types of cases will include a polynomial term.
- Inspection of residuals. If we try to fit a linear model to curved data, a scatter plot of residuals (Y-axis) on the predictor (X-axis) will have patches of many positive residuals in the middle. Hence in such a situation, it is not appropriate.
- An assumption in the usual multiple linear regression analysis is that all the independent variables are independent. In the **polynomial regression** model, this assumption is not satisfied.

How does a Polynomial Regression work?

If we observe closely then we will realize that to evolve from linear regression to polynomial regression. We are just supposed to add the higher-order terms of the dependent features in the feature space. This is sometimes also known as **feature engineering** but not exactly.

Application of Polynomial Regression

The reason behind the vast use cases of the polynomial regression is that approximately all of the real-world data is non-linear in nature and hence when we fit a non-linear model on the data or a curvilinear regression line then the results that we obtain are far better than what we can achieve with the standard linear regression. Some of the use cases of the Polynomial regression are as stated below:

- The growth rate of tissues.
- Progression of disease epidemics
- Distribution of carbon isotopes in lake sediments

The basic goal of regression analysis is to model the expected value of a dependent variable y in terms of the value of an independent variable x. In simple **linear regression**, we used the following equation –

```
y = a + bx + e
```

Here y is a dependent variable, a is the y-intercept, b is the slope and e is the error rate. In many cases, this linear model will not work out For example if we analyze the production of chemical synthesis in terms of the temperature at which the synthesis takes place in such cases we use a quadratic model

```
y = a + b1x + b22 + e
```

Here y is the dependent variable on x, a is the y-intercept and e is the error rate. In general, we can model it for the nth value.

```
y = a + b1x + b2x2 +....+ bnxn
```

Since the regression function is linear in terms of unknown variables, hence these models are linear from the point of estimation. Hence through the **Least Square technique**, let’s compute the response value that is y.

Polynomial Regression in Python

To get the Dataset used for the analysis of Polynomial Regression, click **here**. Import the important libraries and the dataset we are using to perform Polynomial Regression.

Python libraries make it very easy for us to handle the data and perform typical and complex tasks with a single line of code.

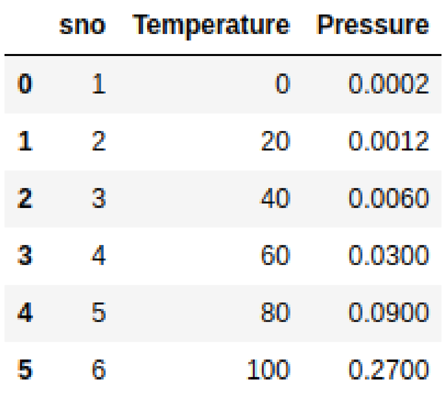
- **Pandas** – This library helps to load the data frame in a 2D array format and has multiple functions to perform analysis tasks in one go.
- **Numpy** – Numpy arrays are very fast and can perform large computations in a very short time.
- **Matplotlib/Seaborn** – This library is used to draw visualizations.
- Sklearn – This module contains multiple libraries having pre-implemented functions to perform tasks from data preprocessing to model development and evaluation.

Python3

```
# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

```
# Importing the dataset
datas = pd.read_csv('data.csv')
datas
```

Output:



	sno	Temperature	Pressure
0	1	0	0.0002
1	2	20	0.0012
2	3	40	0.0060
3	4	60	0.0300
4	5	80	0.0900
5	6	100	0.2700

First Five rows of the dataset

Our feature variable that is **X** will contain the Column between 1st and the target variable that is **y** will contain the 2nd column.

Now let's fit a linear regression model on the data at hand.

```
# Features and the target variables
X = datas.iloc[:, 1:2].values
y = datas.iloc[:, 2].values

# Fitting Linear Regression to the dataset
from sklearn.linear_model import LinearRegression
lin = LinearRegression()

lin.fit(X, y)
```

Fitting the Polynomial Regression model on two components X and y.

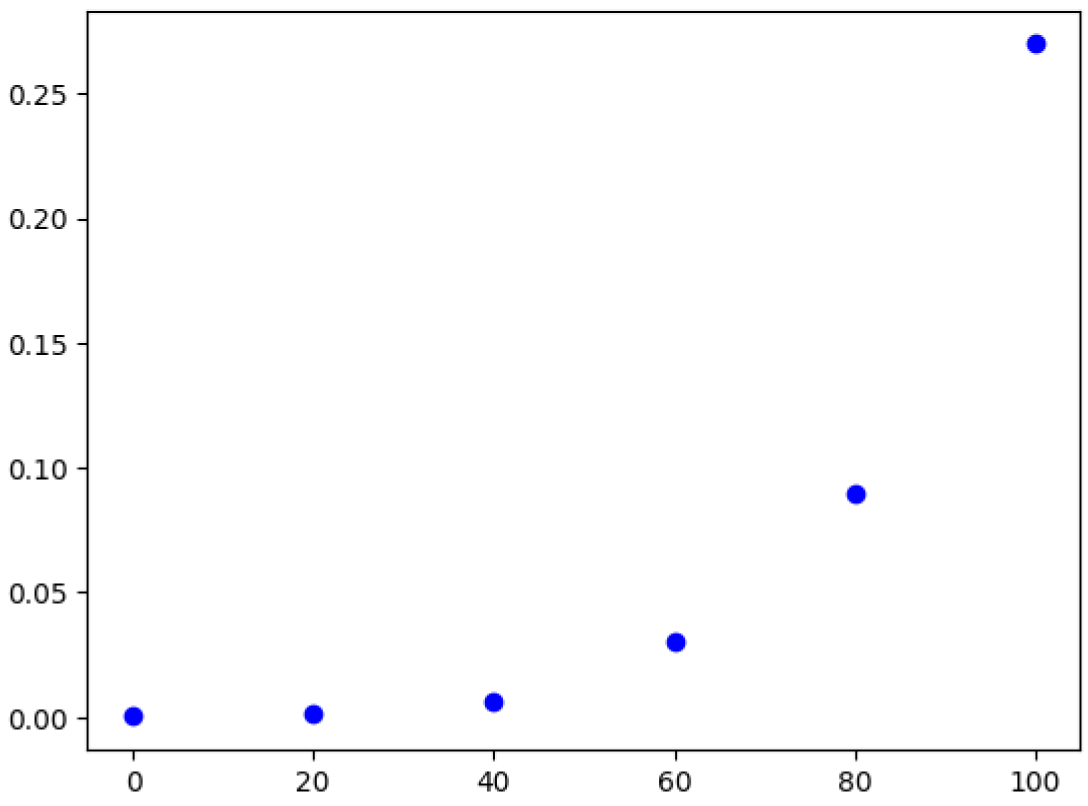
```
# Features and the target variables
X = datas.iloc[:, 1:2].values
y = datas.iloc[:, 2].values
```

```
# Fitting Linear Regression to the dataset
from sklearn.linear_model import LinearRegression
lin = LinearRegression()

lin.fit(X, y)
```

In this step, we are Visualising the Linear Regression results using a scatter plot.

```
# Visualising the Linear Regression results
plt.scatter(X, y, color='blue')

plt.plot(X, lin.predict(X), color='red')
plt.title('Linear Regression')
plt.xlabel('Temperature')
plt.ylabel('Pressure')

plt.show()
```

Output:

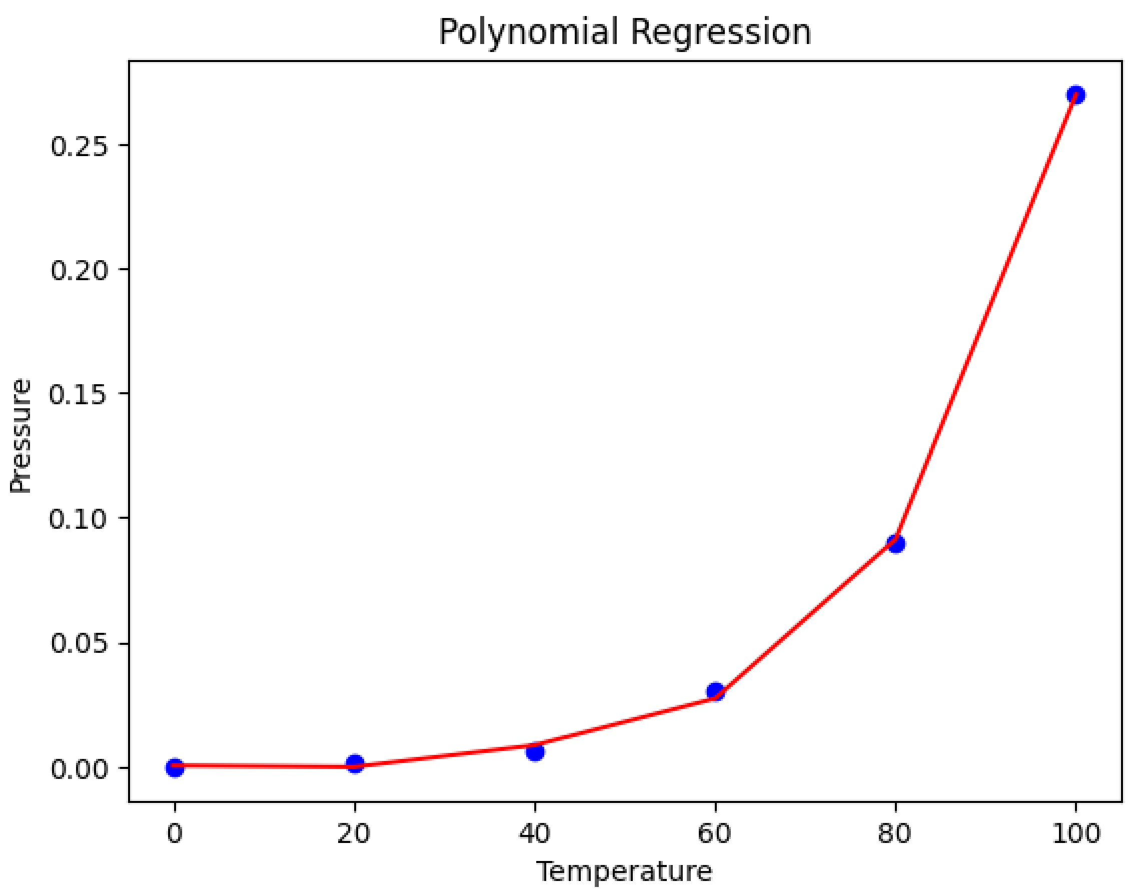


Scatter plot of feature and the target variable.

Visualize the Polynomial Regression results using a scatter plot.

```
# Visualising the Polynomial Regression results
plt.scatter(X, y, color='blue')

plt.plot(X, lin2.predict(poly.fit_transform(X)),
        color='red')
plt.title('Polynomial Regression')
plt.xlabel('Temperature')
plt.ylabel('Pressure')

plt.show()
```

Output:



Implementation of Polynomial Regression

Predict new results with both Linear and Polynomial Regression. Note that the input variable must be in a Numpy 2D array.

```
# Predicting a new result with Linear Regression
# after converting predict variable to 2D array
pred = 110.0
predarray = np.array([[pred]])
lin.predict(predarray)
```

Output:

```
array([0.20675333])
```

```
# Predicting a new result with Polynomial Regression
# after converting predict variable to 2D array
pred2 = 110.0
```

```
pred2array = np.array([[pred2]])
lin2.predict(poly.fit_transform(pred2array))
```

Output:

```
array([0.43295877])
```

Overfitting Vs Under-fitting

While dealing with the polynomial regression one thing that we face is the problem of **overfitting** this happens because while we increase the order of the polynomial regression to achieve better and better performance model gets overfit on the data and does not perform on the new data points.

Due to this reason only while using the polynomial regression, do we try to penalize the weights of the model to regularize the effect of the overfitting problem. **Regularization** techniques like **Lasso regression**and **Ridge regression** methodologies are used whenever we deal with a situation in which the model may overfit the data at hand.

Bias Vs Variance Tradeoff

This technique is the generalization of the approach that is used to avoid the problem of overfitting and underfitting. Here as well this technique helps us to avoid the problem of overfitting by helping us select the appropriate value for the degree of the polynomial we are trying to fit our data on. For example, this is achieved when after increasing the degree of polynomial after a certain level the gap between the training and the validation metrics starts increasing.

Advantages of using Polynomial Regression

- A broad range of functions can be fit under it.
- Polynomial basically fits a wide range of curvatures.
- Polynomial provides the best approximation of the relationship between dependent and independent variables.

Disadvantages of using Polynomial Regression

- These are too sensitive to outliers.

- The presence of one or two **outliers** in the data can seriously affect the results of nonlinear analysis.
- In addition, there are unfortunately fewer model validation tools for the detection of outliers in nonlinear regression than there are for linear regression.

Stepwise Regression

Stepwise regression is used for fitting regression models with predictive models. It is carried out automatically. With each step, the variable is added or subtracted from the set of explanatory variables. The approaches for stepwise regression are forward selection, backward elimination, and bidirectional elimination. The formula for stepwise regression is

$$b_{j.std} = b_j(s_x * s_y^{-1})$$

Stepwise regression is a method of fitting a regression model by iteratively adding or removing variables. It is used to build a model that is accurate and parsimonious, meaning that it has the smallest number of variables that can explain the data.

There are two main types of stepwise regression:

- Forward Selection – In forward selection, the algorithm starts with an empty model and iteratively adds variables to the model until no further improvement is made.
- Backward Elimination – In backward elimination, the algorithm starts with a model that includes all variables and iteratively removes variables until no further improvement is made.

The advantage of stepwise regression is that it can automatically select the most important variables for the model and build a parsimonious model. The disadvantage is that it may not always select the best model, and it can be sensitive to the order in which the variables are added or removed.

Use of Stepwise Regression?

The primary use of stepwise regression is to build a regression model that is accurate and parsimonious. In other words, it is used to find the smallest number of variables that can explain the data.

Stepwise regression is a popular method for model selection because it can automatically select the most important variables for the model and build a parsimonious model. This can save time and effort for the data scientist or analyst, who does not have to manually select the variables for the model.

Stepwise regression can also improve the model's performance by reducing the number of variables and eliminating any unnecessary or irrelevant variables. This can help to prevent overfitting, which can occur when the model is too complex and does not generalize well to new data.

Overall, the use of stepwise regression is to build accurate and parsimonious regression models that can handle complex, non-linear relationships in the data. It is a popular and effective method for model selection in many different domains.

Stepwise Regression And Other Regression Models?

Stepwise regression is different from other regression methods because it automatically selects the most important variables for the model. Other regression methods, such as **ordinary least squares** (OLS) and least absolute shrinkage and selection operator (**LASSO**), require the data scientist or analyst to manually select the variables for the model.

The advantage of stepwise regression is that it can save time and effort for the data scientist or analyst, and it can also improve the model's performance by reducing the number of variables and eliminating any unnecessary or irrelevant variables. The disadvantage is that it may not always select the best model, and it can be sensitive to the order in which the variables are added or removed.

Overall, stepwise regression is a useful method for model selection, but it should be used carefully and in combination with other regression methods to ensure that the best model is selected.

Difference between stepwise regression and Linear regression

Linear regression is a statistical method used to model the relationship between a dependent variable and one or more independent variables by fitting a linear equation to observed data. In other words, it is a method for predicting a response (or dependent variable) based on one or more predictor variables.

Stepwise regression is a method for building a regression model by adding or removing predictors in a step-by-step fashion. The goal of stepwise regression is to identify the subset of predictors that provides the best predictive performance for the response variable. This is done by starting with an empty model and iteratively adding or removing predictors based on the strength of their relationship with the response variable.

In summary, linear regression is a method for modeling the relationship between a response and one or more predictor variables, while stepwise regression is a method for building a regression model by iteratively adding or removing predictors.

Implemplementation of Stepwise Regression in Python

To perform stepwise regression in **Python**, you can follow these steps:

- Install the mlxtend library by running pip install mlxtend in your command prompt or terminal.
- Import the necessary modules from the mlxtend library, including sequential_feature_selector and linear_model.
- Define the features and target variables in your dataset.
- Initialize the stepwise regression model with the sequential_feature_selector and specify the type of regression to be used (e.g. linear_model.LinearRegression for linear regression).
- Fit the stepwise regression model to your dataset using the fit method.

Use the k_features attribute of the fitted model to see which features were selected by the stepwise regression.

Importing Libraries

To implement stepwise regression, you will need to have the following libraries installed:

- **Pandas**: For data manipulation and analysis.
- **NumPy**: For working with arrays and matrices.
- **Sklearn**: for machine learning algorithms and preprocessing tools

- **mlxtend**: for feature selection algorithms. Not mandatory but useful

The first step is to define the array of data and convert it into a dataframe using the NumPy and pandas libraries. Then, the features and target are selected from the dataframe using the **iloc** method.

```
import pandas as pd
import numpy as np
from sklearn import linear_model
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from mlxtend.feature_selection import SequentialFeatureSele
ctor

# Define the array of data
data = np.array([[1, 2, 3, 4],
                [5, 6, 7, 8],
                [9, 10, 11, 12]])

# Convert the array into a dataframe
df = pd.DataFrame(data)

# Select the features and target
X = df.iloc[:, :-1]
y = df.iloc[:, -1]
```

Model Development in Stepwise Regression

Next, stepwise regression is performed using the **SequentialFeatureSelector()** function from the mlxtend library. This function uses a logistic regression model to select the most important features in the dataset, and the number of selected features can be specified using the k_features parameter.

```
# Perform stepwise regression
sfs = SequentialFeatureSelector(linear_model.LogisticRegres
sion(),
                                k_features=3,
                                forward=True,
                                scoring='accuracy',
                                cv=None)
selected_features = sfs.fit(X, y)
```

After the stepwise regression is complete, the selected features are checked using the selected_features.k_feature_names_ attribute and a data frame with only the selected features are created. Finally, the data is split into train and test sets using the **train_test_split()** function from the sklearn library, and a logistic regression model is fit using the selected features. The model performance is then evaluated using the accuracy_score() function from the sklearn library.

```
# Create a dataframe with only the selected features
selected_columns = [0, 1, 2, 3]
df_selected = df[selected_columns]

# Split the data into train and test sets
X_train, X_test,\
    y_train, y_test = train_test_split(
        df_selected, y,
        test_size=0.3,
        random_state=42)

# Fit a logistic regression model using the selected featur
es
logreg = linear_model.LogisticRegression()
logreg.fit(X_train, y_train)

# Make predictions using the test set
y_pred = logreg.predict(X_test)

# Evaluate the model performance
print(y_pred)
```

Output:

```
[8]
```

The difference between linear regression and stepwise regression is that stepwise regression is a method for building a regression model by iteratively adding or removing predictors, while linear regression is a method for modeling the relationship between a response and one or more predictor variables.

In the stepwise regression examples, the mlxtend library is used to iteratively add or remove predictors based on their relationship with the response variable, while in the linear regression examples, all predictors are used to fit the model.

Decision Tree Regression

A Decision Tree is the most powerful and popular tool for classification and prediction. A **Decision tree** is a flowchart-like tree structure, where each internal node denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node (terminal node) holds a class label. There is a non-parametric method used to model a decision tree to predict a continuous outcome.

A decision tree is one of the most powerful tools of supervised learning algorithms used for both classification and regression tasks. It builds a flowchart-like tree structure where each internal node denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node (terminal node) holds a class label. It is constructed by recursively splitting the training data into subsets based on the values of the attributes until a stopping criterion is met, such as the maximum depth of the tree or the minimum number of samples required to split a node.

During training, the Decision Tree algorithm selects the best attribute to split the data based on a metric such as entropy or Gini impurity, which measures the level of impurity or randomness in the subsets. The goal is to find the attribute that maximizes the information gain or the reduction in impurity after the split.

What is a Decision Tree?

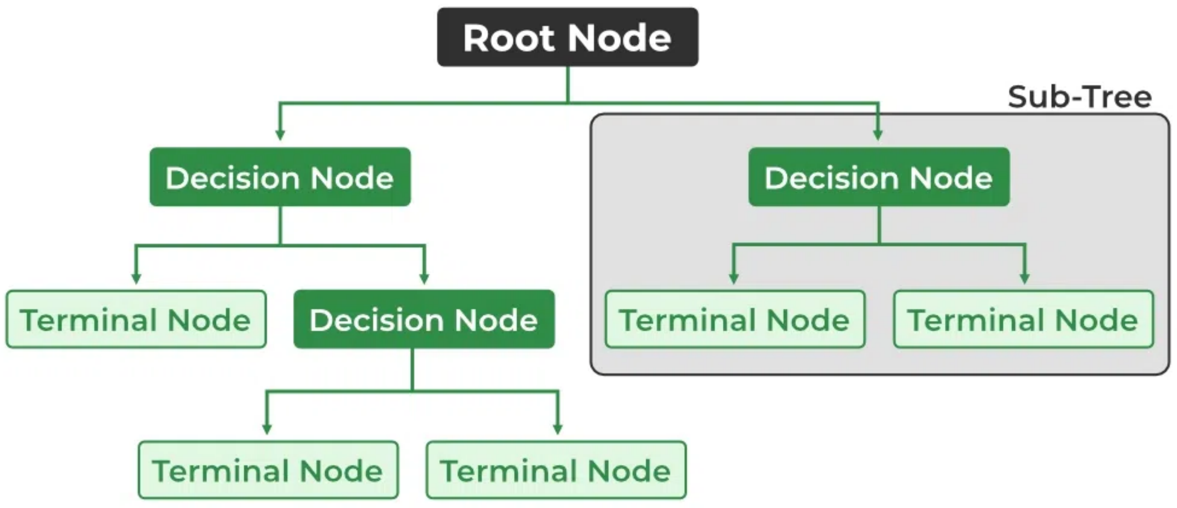
A decision tree is a flowchart-like **tree structure** where each internal node denotes the feature, branches denote the rules and the leaf nodes denote the result of the algorithm. It is a versatile **supervised machine-learning** algorithm, which is used for both classification and regression problems. It is one of the very powerful algorithms. And it is also used in Random Forest to train on different subsets of training data, which makes random forest one of the most powerful algorithms in **machine learning**.

Decision Tree Terminologies

Some of the common Terminologies used in Decision Trees are as follows:

- **Root Node:** It is the topmost node in the tree, which represents the complete dataset. It is the starting point of the decision-making process.
- Decision/Internal Node: A node that symbolizes a choice regarding an input feature. Branching off of internal nodes connects them to leaf nodes or other internal nodes.
- **Leaf/Terminal Node:** A node without any child nodes that indicates a class label or a numerical value.
- **Splitting:** The process of splitting a node into two or more sub-nodes using a split criterion and a selected feature.
- **Branch/Sub-Tree:** A subsection of the decision tree starts at an internal node and ends at the leaf nodes.
- **Parent Node:** The node that divides into one or more child nodes.
- **Child Node:** The nodes that emerge when a parent node is split.
- **Impurity**: A measurement of the target variable's homogeneity in a subset of data. It refers to the degree of randomness or uncertainty in a set of examples. The **Gini index** and **entropy** are two commonly used impurity measurements in decision trees for classifications task
- **Variance**: Variance measures how much the predicted and the target variables vary in different samples of a dataset. It is used for regression problems in decision trees. **Mean squared error, Mean Absolute Error, friedman_mse, or Half Poisson deviance** are used to measure the variance for the regression tasks in the decision tree.
- **Information Gain:** Information gain is a measure of the reduction in impurity achieved by splitting a dataset on a particular feature in a decision tree. The splitting criterion is determined by the feature that offers the greatest information gain, It is used to determine the most informative feature to split on at each node of the tree, with the goal of creating pure subsets

- **Pruning**: The process of removing branches from the tree that do not provide any additional information or lead to overfitting.



Decision Tree

Attribute Selection Measures:

Construction of Decision Tree: A tree can be *"learned"* by splitting the source set into subsets based on Attribute Selection Measures. Attribute selection measure (ASM) is a criterion used in decision tree algorithms to evaluate the usefulness of different attributes for splitting a dataset. The goal of ASM is to identify the attribute that will create the most homogeneous subsets of data after the split, thereby maximizing the information gain. This process is repeated on each derived subset in a recursive manner called *recursive partitioning*. The recursion is completed when the subset at a node all has the same value of the target variable, or when splitting no longer adds value to the predictions. The construction of a decision tree classifier does not require any domain knowledge or parameter setting and therefore is appropriate for exploratory knowledge discovery. Decision trees can handle high-dimensional data.

Entropy:

Entropy is the measure of the degree of randomness or uncertainty in the dataset. In the case of classifications, It measures the randomness based on the distribution of class labels in the dataset.

The entropy for a subset of the original dataset having K number of classes for the ith node can be defined as:

$$H_i = -\sum_{k \epsilon K}^{n} p(i,k) \log_2 p(i,k)$$

Where,

- S is the dataset sample.
- k is the particular class from K classes
- p(k) is the proportion of the data points that belong to class k to the total number of data points in dataset sample S.

$$p(k) = \frac{1}{n} \sum I(y=k)$$

- Here p(i,k) should not be equal to zero.

Important points related to Entropy:

1. The entropy is 0 when the dataset is completely homogeneous, meaning that each instance belongs to the same class. It is the lowest entropy indicating no uncertainty in the dataset sample.
2. when the dataset is equally divided between multiple classes, the entropy is at its maximum value. Therefore, entropy is highest when the distribution of class labels is even, indicating maximum uncertainty in the dataset sample.
3. Entropy is used to evaluate the quality of a split. The goal of entropy is to select the attribute that minimizes the entropy of the resulting subsets, by splitting the dataset into more homogeneous subsets with respect to the class labels.
4. The highest information gain attribute is chosen as the splitting criterion (i.e., the reduction in entropy after splitting on that attribute), and the process is repeated recursively to build the decision tree.

Gini Impurity or index:

Gini Impurity is a score that evaluates how accurate a split is among the classified groups. The Gini Impurity evaluates a score in the range between 0 and 1, where 0 is when all observations belong to one class, and 1 is a random distribution of the elements within classes. In this case, we want to have a Gini index score as low as possible. Gini Index is the evaluation metric we shall use to evaluate our Decision Tree Model.

$$\text{Gini Impurity} = 1 - \sum p_i^2$$

Information Gain:

Information gain measures the reduction in entropy or variance that results from splitting a dataset based on a specific property. It is used in decision tree algorithms to determine the usefulness of a feature by partitioning the dataset into more homogeneous subsets with respect to the class labels or target variable. The higher the information gain, the more valuable the feature is in predicting the target variable.

The information gain of an attribute A, with respect to a dataset S, is calculated as follows:

$$\text{Information Gain(H, A)} = H - \sum \frac{|H_V|}{|H|} H_v$$

where

- A is the specific attribute or class label
- |H| is the entropy of dataset sample S
- |Hv| is the number of instances in the subset S that have the value v for attribute A

Information gain measures the reduction in entropy or variance achieved by partitioning the dataset on attribute A. The attribute that maximizes information gain is chosen as the splitting criterion for building the decision tree.

Information gain is used in both classification and regression decision trees. In classification, entropy is used as a measure of impurity, while in regression, variance is used as a measure of impurity. The information gain calculation remains the same in both cases, except that entropy or variance is used instead of entropy in the formula.

How does the Decision Tree algorithm Work?

The decision tree operates by analyzing the data set to predict its classification. It commences from the tree's root node, where the algorithm views the value of the root attribute compared to the attribute of the record in the actual data set. Based on the comparison, it proceeds to follow the branch and move to the next node.

The algorithm repeats this action for every subsequent node by comparing its attribute values with those of the sub-nodes and continuing the process further. It repeats until it reaches the leaf node of the tree. The complete mechanism can be better explained through the algorithm given below.

- Step-1: Begin the tree with the root node, says S, which contains the complete dataset.
- Step-2: Find the best attribute in the dataset using Attribute Selection Measure (ASM).
- Step-3: Divide the S into subsets that contains possible values for the best attributes.
- Step-4: Generate the decision tree node, which contains the best attribute.
- Step-5: Recursively make new decision trees using the subsets of the dataset created in step -3. Continue this process until a stage is reached where you cannot further classify the nodes and called the final node as a leaf nodeClassification and Regression Tree algorithm.

Advantages of the Decision Tree:

1. It is simple to understand as it follows the same process which a human follow while making any decision in real-life.
2. It can be very useful for solving decision-related problems.
3. It helps to think about all the possible outcomes for a problem.
4. There is less requirement of data cleaning compared to other algorithms.

Disadvantages of the Decision Tree:

1. The decision tree contains lots of layers, which makes it complex.
2. It may have an overfitting issue, which can be resolved using the Random Forest algorithm.
3. For more class labels, the computational complexity of the decision tree may increase.

What are appropriate problems for Decision tree learning?

Although a variety of decision tree learning methods have been developed with somewhat differing capabilities and requirements, decision tree learning is generally best suited to problems with the following characteristics:

1. Instances are represented by attribute-value pairs:

In the world of decision tree learning, we commonly use attribute-value pairs to represent instances. An instance is defined by a predetermined group of attributes, such as temperature, and its corresponding value, such as hot. Ideally, we want each attribute to have a finite set of distinct values, like hot, mild, or cold. This makes it easy to construct decision trees. However, more advanced versions of the algorithm can accommodate attributes with continuous numerical values, such as representing temperature with a numerical scale.

2. The target function has discrete output values:

The marked objective has distinct outcomes. The decision tree method is ordinarily employed for categorizing Boolean examples, such as yes or no. Decision tree approaches can be readily expanded for acquiring functions with beyond dual conceivable outcome values. A more substantial expansion lets us gain knowledge about aimed objectives with numeric outputs, although the practice of decision trees in this framework is comparatively rare.

3. Disjunctive descriptions may be required:

Decision trees naturally represent disjunctive expressions.

4.The training data may contain errors:

"Techniques of decision tree learning demonstrate high resilience towards discrepancies, including inconsistencies in categorization of sample cases and discrepancies in the feature details that characterize these cases."

5. The training data may contain missing attribute values:

In certain cases, the input information designed for training might have absent characteristics. Employing decision tree approaches can still be possible despite experiencing unknown features in some training samples. For instance, when considering the level of humidity throughout the day, this information may only be accessible for a specific set of training specimens.

Practical issues in learning decision trees include:

- Determining how deeply to grow the decision tree,

- Handling continuous attributes,
- Choosing an appropriate attribute selection measure,
- Handling training data with missing attribute values,
- Handling attributes with differing costs, and
- Improving computational efficiency.
-

To build the Decision Tree, **CART (Classification and Regression Tree) algorithm** is used. It works by selecting the best split at each node based on metrics like Gini impurity or information Gain. In order to create a decision tree. Here are the basic steps of the CART algorithm:

1. The root node of the tree is supposed to be the complete training dataset.
2. Determine the impurity of the data based on each feature present in the dataset. Impurity can be measured using metrics like the Gini index or entropy for classification and Mean squared error, Mean Absolute Error, friedman_mse, or Half Poisson deviance for regression.
3. Then selects the feature that results in the highest information gain or impurity reduction when splitting the data.
4. For each possible value of the selected feature, split the dataset into two subsets (left and right), one where the feature takes on that value, and another where it does not. The split should be designed to create subsets that are as pure as possible with respect to the target variable.
5. Based on the target variable, determine the impurity of each resulting subset.
6. For each subset, repeat steps 2–5 iteratively until a stopping condition is met. For example, the stopping condition could be a maximum tree depth, a minimum number of samples required to make a split or a minimum impurity threshold.
7. Assign the majority class label for classification tasks or the mean value for regression tasks for each terminal node (leaf node) in the tree.

Classification and Regression Tree algorithm for Classification

Let the data available at node m be Qm and it has nm samples. and tm as the threshold for node m. then, The classification and regression tree algorithm for classification can be written as :

$$G(Q_m, t_m) = \frac{n_m^{Left}}{n_m} H(Q_m^{Left}(t_m)) + \frac{n_m^{Right}}{n_m} H(Q_m^{Right}(t_m))$$

Here,

- H is the measure of impurities of the left and right subsets at node m. it can be entropy or Gini impurity.
- n is the number of instances in the left and right subsets at node m.

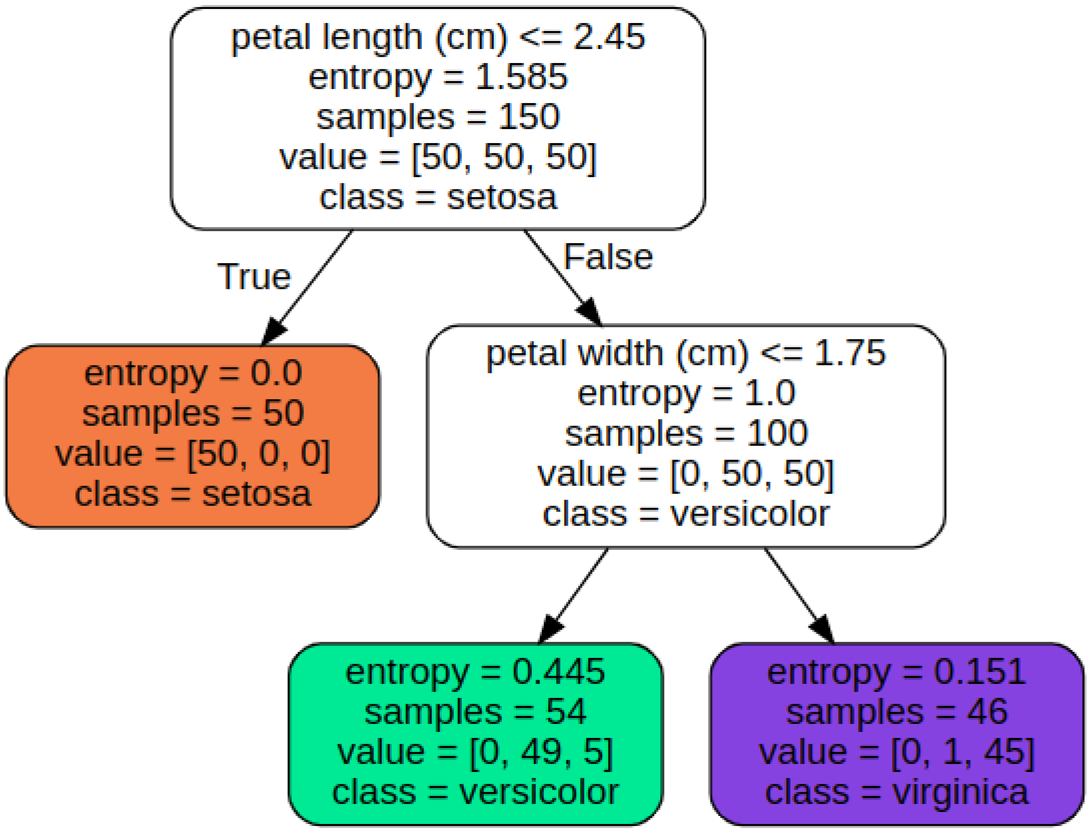
  m

To select the parameter, we can write as:

$$t_m =_{t_m} H(Q_m, t_m)$$

```
# Import the necessary libraries
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import export_graphviz
from graphviz import Source

# Load the dataset
iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target

# DecisionTreeClassifier
tree_clf = DecisionTreeClassifier(criterion='entropy',
                                max_depth=2)
tree_clf.fit(X, y)

# Plot the decision tree graph
export_graphviz(
    tree_clf,
    out_file="iris_tree.dot",
    feature_names=iris.feature_names[2:],
    class_names=iris.target_names,
    rounded=True,
    filled=True
)

with open("iris_tree.dot") as f:
    dot_graph = f.read()

Source(dot_graph)
```

Output:



Classification and Regression Tree algorithm for Regression

Let the data available at node m be Qm and it has nm samples. and tm as the threshold for node m. then, The classification and regression tree algorithm for regression can be written as :

$$G(Q_m, t_m) = \frac{n_m^{Left}}{n_m} MSE(Q_m^{Left}(t_m)) + \frac{n_m^{Right}}{n_m} MSE(Q_m^{Right}(t_m))$$

Here,

- MSE is the mean squared error.

$$MSE_{Q_m} = \sum_{y \epsilon Q_m} (\bar{y}_m - y)^2$$
$$\text{where, } \bar{y}_m = \frac{1}{n_m} \sum_{y \epsilon Q_m} y$$

- n is the number of instances in the left and right subsets at node m.

  m

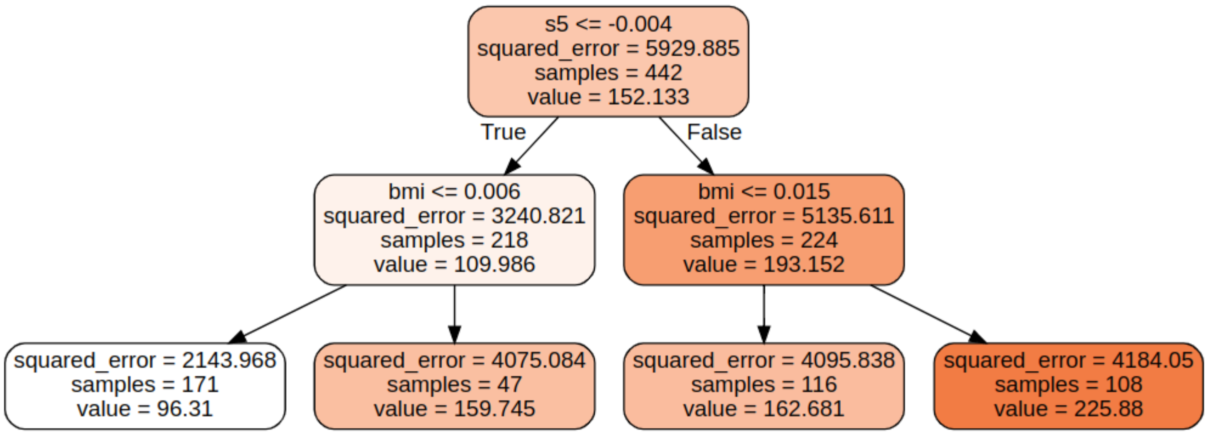
To select the parameter, we can write as:

$$t_m =_{t_m} MSE(Q_m, t_m)$$

Example:

```
# Import the necessary libraries
from sklearn.datasets import load_diabetes
from sklearn.tree import DecisionTreeRegressor
from sklearn.tree import export_graphviz
from graphviz import Source

# Load the dataset
diabetes = load_diabetes()
X = diabetes.data
y = diabetes.target

# DecisionTreeRegressor
tree_reg = DecisionTreeRegressor(criterion = 'squared_erro
r',
                                max_depth=2)

tree_reg.fit(X, y)

# Plot the decision tree graph
export_graphviz(
    tree_reg,
    out_file="diabetes_tree.dot",
    feature_names=diabetes.feature_names,
    class_names=diabetes.target,
    rounded=True,
    filled=True
)

with open("diabetes_tree.dot") as f:
    dot_graph = f.read()

Source(dot_graph)
```

Output:



Decision Tree Regression

Strengths and Weaknesses of the Decision Tree Approach

The strengths of decision tree methods are:

- Decision trees are able to generate understandable rules.
- Decision trees perform classification without requiring much computation.
- Decision trees are able to handle both continuous and categorical variables.
- Decision trees provide a clear indication of which fields are most important for prediction or classification.
- Ease of use: Decision trees are simple to use and don't require a lot of technical expertise, making them accessible to a wide range of users.
- Scalability: Decision trees can handle large datasets and can be easily parallelized to improve processing time.

- Missing value tolerance: Decision trees are able to handle missing values in the data, making them a suitable choice for datasets with missing or incomplete data.
- Handling non-linear relationships: Decision trees can handle non-linear relationships between variables, making them a suitable choice for complex datasets.
- Ability to handle imbalanced data: Decision trees can handle imbalanced datasets, where one class is heavily represented compared to the others, by weighting the importance of individual nodes based on the class distribution.

The weaknesses of decision tree methods :

- Decision trees are less appropriate for estimation tasks where the goal is to predict the value of a continuous attribute.
- Decision trees are prone to errors in classification problems with many classes and a relatively small number of training examples.
- Decision trees can be computationally expensive to train. The process of growing a decision tree is computationally expensive. At each node, each candidate splitting field must be sorted before its best split can be found. In some algorithms, combinations of fields are used and a search must be made for optimal combining weights. Pruning algorithms can also be expensive since many candidate sub-trees must be formed and compared.
- Decision trees are prone to overfitting the training data, particularly when the tree is very deep or complex. This can result in poor performance on new, unseen data.
- Small variations in the training data can result in different decision trees being generated, which can be a problem when trying to compare or reproduce results.
- Many decision tree algorithms do not handle missing data well, and require imputation or deletion of records with missing values.
- The initial splitting criteria used in decision tree algorithms can lead to biased trees, particularly when dealing with unbalanced datasets or rare classes.
- Decision trees are limited in their ability to represent complex relationships between variables, particularly when dealing with nonlinear or interactive

effects.

- Decision trees can be sensitive to the scaling of input features, particularly when using distance-based metrics or decision rules that rely on comparisons between values.

```
from sklearn.datasets import make_classification
from sklearn import tree
from sklearn.model_selection import train_test_split

X, t = make_classification(100, 5, n_classes=2, shuffle=Tru
e, random_state=10)
X_train, X_test, t_train, t_test = train_test_split(
    X, t, test_size=0.3, shuffle=True, random_state=1)

model = tree.DecisionTreeClassifier()
model = model.fit(X_train, t_train)

predicted_value = model.predict(X_test)
print(predicted_value)

tree.plot_tree(model)

zeroes = 0
ones = 0
for i in range(0, len(t_train)):
    if t_train[i] == 0:
        zeroes += 1
    else:
        ones += 1

print(zeroes)
print(ones)

val = 1 - ((zeroes/70)*(zeroes/70) + (ones/70)*(ones/70))
print("Gini :", val)

match = 0
UnMatch = 0
```

```
for i in range(30):
    if predicted_value[i] == t_test[i]:
        match += 1
    else:
        UnMatch += 1

accuracy = match/30
print("Accuracy is: ", accuracy)
```

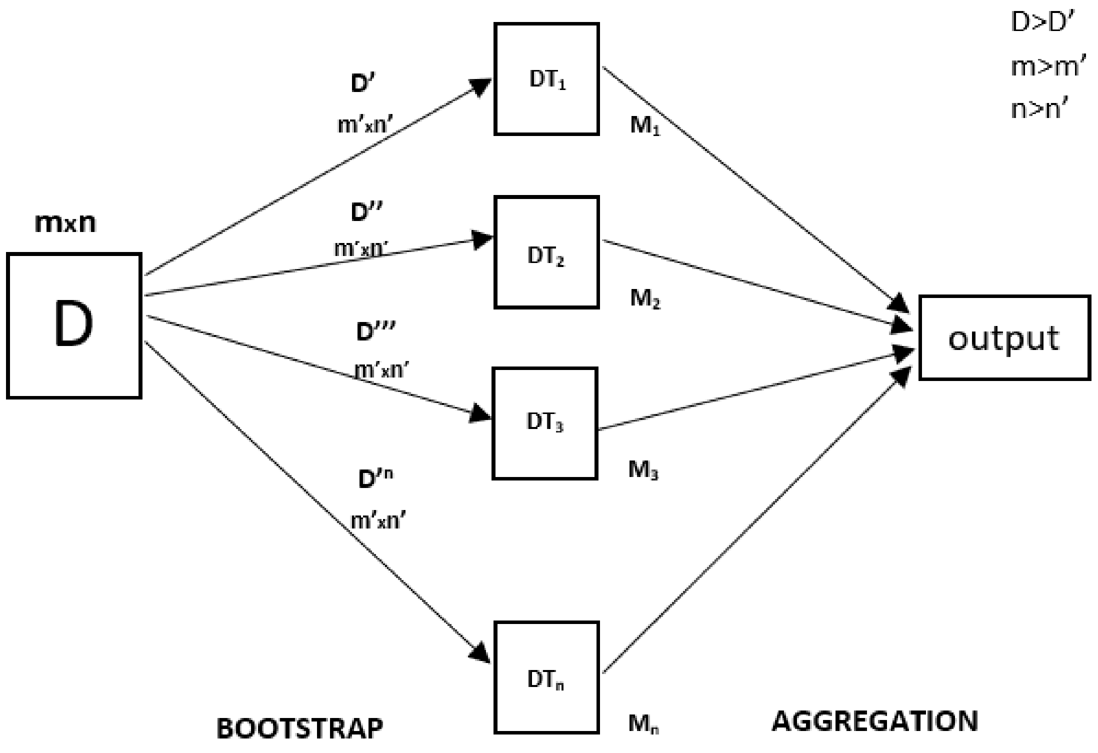
```
1 1 0 0 1 0 1 0 1 0 0 1 1 0 0 1 0 1 1 1 0 0 0 1 1 0 0 0 1 0
Gini : 0.5
Accuracy is: 0.366667
```

Random Forest Regression

Random Forest is an **ensemble** technique capable of performing both regression and classification tasks with the use of multiple decision trees and a technique called Bootstrap and Aggregation, commonly known as **bagging**. The basic idea behind this is to combine multiple decision trees in determining the final output rather than relying on individual decision trees.

Random Forest has multiple decision trees as base learning models. We randomly perform row sampling and feature sampling from the dataset forming sample datasets for every model. This part is called Bootstrap.

Every decision tree has high variance, but when we combine all of them together in parallel then the resultant variance is low as each decision tree gets perfectly trained on that particular sample data, and hence the output doesn’t depend on one decision tree but on multiple decision trees. In the case of a classification problem, the final output is taken by using the majority voting classifier. In the case of a regression problem, the final output is the mean of all the outputs. This part is called **Aggregation**.



Random Forest Regression Model Working

What is Random Forest Regression?

Random Forest is an **ensemble technique** capable of performing both **regression and classification** tasks with the use of multiple decision trees and a technique called Bootstrap and Aggregation, commonly known as **bagging**. The basic idea behind this is to combine multiple decision trees in determining the final output rather than relying on individual decision trees.

Random Forest has multiple decision trees as base learning models. We randomly perform row sampling and feature sampling from the dataset forming sample datasets for every model. This part is called Bootstrap.

We need to approach the Random Forest regression technique like any other **machine learning** technique.

- Design a specific question or data and get the source to determine the required data.
- Make sure the data is in an accessible format else convert it to the required format.

- Specify all noticeable anomalies and missing data points that may be required to achieve the required data.
- Create a machine-learning model.
- Set the baseline model that you want to achieve
- Train the data machine learning model.
- Provide an insight into the model with test data
- Now compare the performance metrics of both the test data and the predicted data from the model.
- If it doesn't satisfy your expectations, you can try improving your model accordingly or dating your data, or using another data modeling technique.
- At this stage, you interpret the data you have gained and report accordingly.

You will be using a similar sample technique in the below example. Below is a step-by-step sample implementation of Random Forest Regression, on the dataset that can be downloaded here- https://bit.ly/417n3N5

Import Libraries and Datasets

Python libraries make it very easy for us to handle the data and perform typical and complex tasks with a single line of code.

- **Pandas** – This library helps to load the data frame in a 2D array format and has multiple functions to perform analysis tasks in one go.
- **Numpy** – Numpy arrays are very fast and can perform large computations in a very short time.
- **Matplotlib/Seaborn** – This library is used to draw visualizations.
- Sklearn – This module contains multiple libraries having pre-implemented functions to perform tasks from data preprocessing to model development and evaluation.
- **RandomForestRegressor** – This is the regression model that is based upon the Random Forest model or the ensemble learning that we will be using in this article using the sklearn library.

```
# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

Now let's load the dataset in the panda's data frame. For better data handling and leveraging the handy functions to perform complex tasks in one go.

```
data = pd.read_csv('Salaries.csv')
print(data)
```

Output:

```
            Position  Level   Salary
0   Business Analyst      1    45000
1  Junior Consultant      2    50000
2  Senior Consultant      3    60000
3            Manager      4    80000
4    Country Manager      5   110000
5     Region Manager      6   150000
6            Partner      7   200000
7     Senior Partner      8   300000
8            C-level      9   500000
9                CEO     10  1000000
```

Select all rows and column 1 from the dataset to x and all rows and column 2 as y.

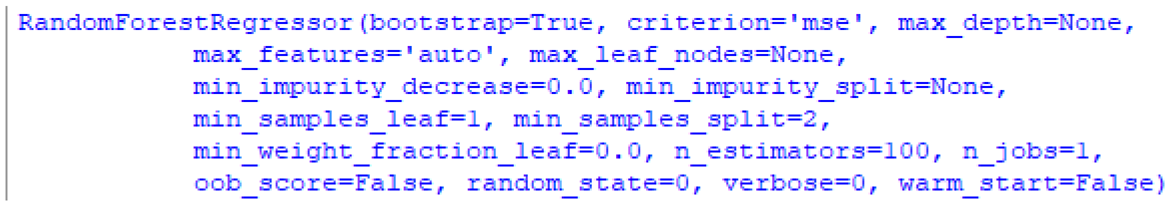
x = df.iloc[:, : -1]
y = df.iloc[:, -1:]

The function enables us to select a particular cell of the dataset, that is, it helps us select a value that belongs to a particular row or column from a set of values of a data frame or dataset.

Random Forest Regressor Model

```
# Fitting Random Forest Regression to the dataset
# import the regressor
from sklearn.ensemble import RandomForestRegressor

# create regressor object
regressor = RandomForestRegressor(n_estimators=100,
                                  random_state=0)

# fit the regressor with x and y data
regressor.fit(x, y)
```

Output:



```
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
           max_features='auto', max_leaf_nodes=None,
           min_impurity_decrease=0.0, min_impurity_split=None,
           min_samples_leaf=1, min_samples_split=2,
           min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=1,
           oob_score=False, random_state=0, verbose=0, warm_start=False)
```

Predicting a new result.

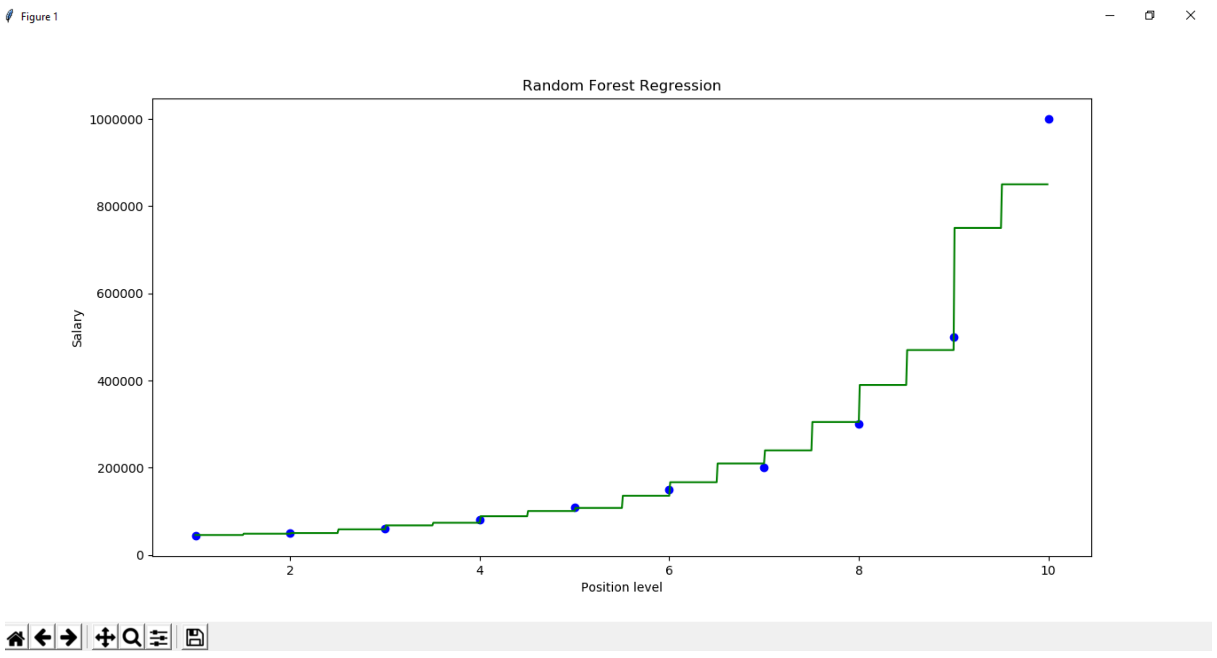
```
# test the output by changing values
Y_pred = regressor.predict(np.array([6.5]).reshape(1, 1))
```

Now let's visualize the results obtained by using the RandomForest Regression model on our salaries dataset.

```
# Visualising the Random Forest Regression results

# arrange for creating a range of values
# from min value of x to max
# value of x with a difference of 0.01
# between two consecutive values
X_grid = np.arrange(min(x), max(x), 0.01)

# reshape for reshaping the data
# into a len(X_grid)*1 array,
# i.e. to make a column out of the X_grid value
```

```
X_grid = X_grid.reshape((len(X_grid), 1))

# Scatter plot for original data
plt.scatter(x, y, color='blue')

# plot predicted data
plt.plot(X_grid, regressor.predict(X_grid),
        color='green')
plt.title('Random Forest Regression')
plt.xlabel('Position level')
plt.ylabel('Salary')
plt.show()
```

Output:



Out of Bag Score in RandomForest

Bag score or **OOB score** is the type of validation technique that is mainly used in bagging algorithms to validate the bagging algorithm. Here a small part of the validation data is taken from the mainstream of the data and the predictions on the particular validation data are done and compared with the other results.

The main advantage that the OOB score offers is that here the validation data is not seen by the bagging algorithm and that is why the results on the OOB score are the true results that indicated the actual performance of the bagging algorithm.

To get the OOB score of the particular Random Forest algorithm, one needs to set the value “True” for the OOB_Score parameter in the algorithm.

```
from sklearn.trees import RandomForestClassifier
RandomeForest = RandomForestClassifier(oob_score=True)
RandomForest.fit(X_train,y_train)
print(RandomForest.oob_score_)
```

Advantages Random Forest Regression

- It is easy to use and less sensitive to the training data compared to the decision tree.
- It is more accurate than the **decision tree** algorithm.
- It is effective in handling large datasets that have many attributes.
- It can handle missing data, **outliers**, and noisy features.

Disadvantages Random Forest Regression

- The model can also be difficult to interpret.
- This algorithm may require some domain expertise to choose the appropriate parameters like the number of decision trees, the maximum depth of each tree, and the number of features to consider at each split.
- It is computationally expensive, especially for large datasets.
- It may suffer from **overfitting** if the model is too complex or the number of decision trees is too high.

Ridge Regression

Ridge regression is a technique for analyzing multiple regression data. When multicollinearity occurs, least squares estimates are unbiased. This is a regularized linear regression model, it tries to reduce the model complexity by adding a penalty term to the cost function. A degree of bias is added to the regression estimates, and as a result, ridge regression reduces the standard errors.

$$\text{Cost} = \underset{\beta \in \mathbb{R}}{\operatorname{argmin}} \|i - X\beta\|^2 + \lambda \|\beta\|^2$$

A Ridge regressor is basically a regularized version of a Linear Regressor. i.e to the original cost function of linear regressor we add a regularized term that forces the learning algorithm to fit the data and helps to keep the weights lower as possible. The regularized term has the parameter 'alpha' which controls the regularization of the model i.e helps in reducing the variance of the estimates.

Cost Function for Ridge Regressor.

$$J(\Theta) = \frac{1}{m}(X\Theta - Y)^2 + \alpha\frac{1}{2}(\Theta)^2$$

Here,

The first term is our basic linear regression's cost function and the second term is our new regularized weights term which uses the L2 norm to fit the data. If the 'alpha' is zero the model is the same as linear regression and the larger 'alpha' value specifies a stronger regularization.

Note: Before using Ridge regressor it is necessary to scale the inputs, because this model is sensitive to scaling of inputs. So performing the scaling through sklearn's StandardScalar will be beneficial.

```
# importing libraries
from sklearn.linear_model import Ridge
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_boston
from sklearn.preprocessing import StandardScaler

# loading boston dataset
boston = load_boston()
X = boston.data[:, :13]
y = boston.target

print ("Boston dataset keys : \n", boston.keys())

print ("\nBoston data : \n", boston.data)
```

```
# scaling the inputs
scaler = StandardScaler()
scaled_X = scaler.fit_transform(X)

# Train Test split will be used for both models
X_train, X_test, y_train, y_test = train_test_split(scaled_
X, y,
                                                      test_si
ze = 0.3)

# training model with 0.5 alpha value
model = Ridge(alpha = 0.5, normalize = False, tol = 0.001,
\
            solver ='auto', random_state = 42)
model.fit(X_train, y_train)

# predicting the y_test
y_pred = model.predict(X_test)

# finding score for our model
score = model.score(X_test, y_test)
print("\n\nModel score : ", score)
```

Output :

```
Boston dataset keys :
 dict_keys(['feature_names', 'DESCR', 'data', 'target'])

Boston data :
 [[6.3200e-03 1.8000e+01 2.3100e+00 ... 1.5300e+01 3.9690e+
02 4.9800e+00]
 [2.7310e-02 0.0000e+00 7.0700e+00 ... 1.7800e+01 3.9690e+0
2 9.1400e+00]
 [2.7290e-02 0.0000e+00 7.0700e+00 ... 1.7800e+01 3.9283e+0
2 4.0300e+00]
 ...
 [6.0760e-02 0.0000e+00 1.1930e+01 ... 2.1000e+01 3.9690e+0
2 5.6400e+00]
```

```
 [1.0959e-01 0.0000e+00 1.1930e+01 ... 2.1000e+01 3.9345e+0
2 6.4800e+00]
 [4.7410e-02 0.0000e+00 1.1930e+01 ... 2.1000e+01 3.9690e+0
2 7.8800e+00]]


Model score :  0.6819292026260749
```

A newer version RidgeCV comes with built-in Cross-Validation for an alpha, so definitely better. Only pass the array of some alpha range values and it'll automatically choose the optimal value for 'alpha'.

Note : 'tol' is the parameter which measures the loss drop and ensures to stop the model at that provided value position or drop at(global minima value).

Lasso Regression

Lasso regression is a regression analysis method that performs both variable selection and **regularization**. Lasso regression uses soft thresholding. Lasso regression selects only a subset of the provided covariates for use in the final model.

This is another regularized linear regression model, it works by adding a penalty term to the cost function, but it tends to zero out some features' coefficients, which makes it useful for feature selection.

Prerequisites:

1. Linear Regression
2. Gradient Descent

Introduction:

Lasso Regression is also another linear model derived from Linear Regression which shares the same hypothetical function for prediction. The cost function of Linear Regression is represented by J.

$$\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} - h\left(x^{(i)}\right)\right)^2$$

```
Here,m is the total number of training examples in the data
set.
h(x(i)) represents the hypothetical function for predictio
n.
y(i)represents the value of target variable for ith trainin
g example.
```

Linear Regression model considers all the features equally relevant for prediction. When there are many features in the dataset and even some of them are not relevant for the predictive model. This makes the model more complex with a too inaccurate prediction on the test set (or overfitting). Such a model with high variance does not generalize on the new data. So, Lasso Regression comes for the rescue. It introduced an L1 penalty (or equal to the absolute value of the magnitude of weights) in the cost function of Linear Regression. The modified cost function for Lasso Regression is given below.

$$\frac{1}{m}\left[\sum_{i=1}^{m}\left(y^{(i)}-h\left(x^{(i)}\right)\right)^{2}+\lambda\sum_{j=1}^{n}w_{j}\right]$$

```
Here,w(j) represents the weight for jth feature.
n is the number of features in the dataset.
lambda is the regularization strength.
```

Lasso Regression performs both, variable selection and regularization too.

Mathematical Intuition:

During gradient descent optimization, added l1 penalty shrunk weights close to zero or zero. Those weights which are shrunken to zero eliminates the features present in the hypothetical function. Due to this, irrelevant features don’t participate in the predictive model. This penalization of weights makes the hypothesis more simple which encourages the sparsity (model with few parameters).

If the intercept is added, it remains unchanged.

We can control the strength of regularization by hyperparameter lambda. All weights are reduced by the same factor lambda.

Different cases for tuning values of lambda.

1. If lambda is set to be 0, Lasso Regression equals Linear Regression.
2. If lambda is set to be infinity, all weights are shrunk to zero.

If we increase lambda, bias increases if we decrease the lambda variance increase. As lambda increases, more and more weights are shrunk to zero and eliminates features from the model.

Implementation

Dataset used in this implementation can be downloaded from the **link**.

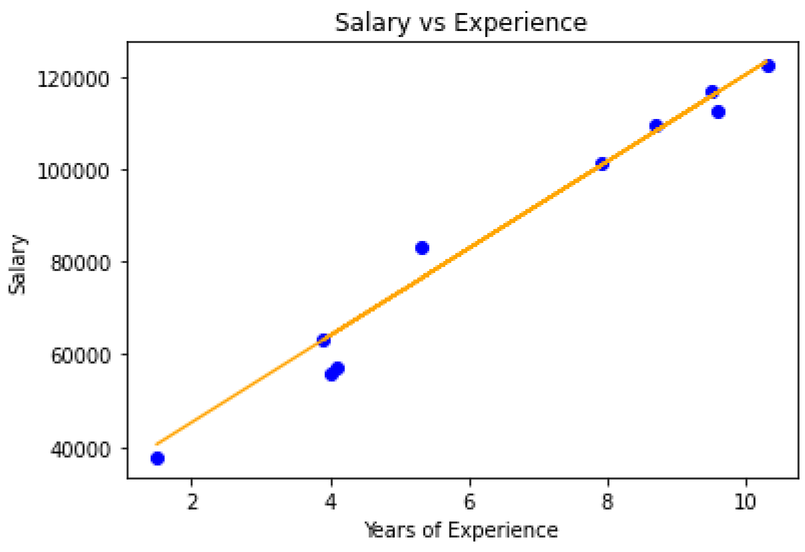
It has 2 columns — "*YearsExperience*" and "*Salary*" for 30 employees in a company. So in this, we will train a Lasso Regression model to learn the correlation between the number of years of experience of each employee and their respective salary. Once the model is trained, we will be able to predict the salary of an employee on the basis of his years of experience.

```
# Importing libraries

import numpy as np

import pandas as pd

from sklearn.model_selection import train_test_split

import matplotlib.pyplot as plt

# Lasso Regression

class LassoRegression() :

    def __init__( self, learning_rate, iterations, l1_penal
ity ) :

        self.learning_rate = learning_rate

        self.iterations = iterations

        self.l1_penality = l1_penality
    # Function for model training

    def fit( self, X, Y ) :

        # no_of_training_examples, no_of_features

        self.m, self.n = X.shape

        # weight initialization

        self.W = np.zeros( self.n )

        self.b = 0

        self.X = X

        self.Y = Y

        # gradient descent learning

        for i in range( self.iterations ) :

            self.update_weights()

        return self

    # Helper function to update weights in gradient descent

    def update_weights( self ) :

        Y_pred = self.predict( self.X )

        # calculate gradients

        dW = np.zeros( self.n )

        for j in range( self.n ) :
            if self.W[j] > 0 :

                dW[j] = ( - ( 2 * ( self.X[:, j] ).dot( sel
f.Y - Y_pred ) )

                        + self.l1_penality ) / self.m

            else :

                dW[j] = ( - ( 2 * ( self.X[:, j] ).dot( sel
f.Y - Y_pred ) )

                        - self.l1_penality ) / self.m


        db = - 2 * np.sum( self.Y - Y_pred ) / self.m

        # update weights

        self.W = self.W - self.learning_rate * dW

        self.b = self.b - self.learning_rate * db

        return self

    # Hypothetical function h( x )

    def predict( self, X ) :

        return X.dot( self.W ) + self.b


def main() :

    # Importing dataset

    df = pd.read_csv( "salary_data.csv" )
    X = df.iloc[:, :-1].values

    Y = df.iloc[:, 1].values

    # Splitting dataset into train and test set

    X_train, X_test, Y_train, Y_test = train_test_split( X,
Y, test_size = 1 / 3, random_state = 0 )

    # Model training

    model = LassoRegression( iterations = 1000, learning_ra
te = 0.01, l1_penality = 500 )

    model.fit( X_train, Y_train )

    # Prediction on test set

    Y_pred = model.predict( X_test )

    print( "Predicted values ", np.round( Y_pred[:3], 2 ) )

    print( "Real values  ", Y_test[:3] )

    print( "Trained W    ", round( model.W[0], 2 ) )

    print( "Trained b    ", round( model.b, 2 ) )

    # Visualization on test set

    plt.scatter( X_test, Y_test, color = 'blue' )

    plt.plot( X_test, Y_pred, color = 'orange' )

    plt.title( 'Salary vs Experience' )

    plt.xlabel( 'Years of Experience' )
    plt.ylabel( 'Salary' )

    plt.show()


if __name__ == "__main__" :

    main()
```

Output:

```
Predicted values  [ 40600.91 123294.39  65033.07]
Real values       [ 37731 122391  57081]
Trained W         9396.99
Trained b         26505.43
```

Visualization



Note: It automates certain parts of model selection and sometimes called variables eliminator.

ElasticNet Regression

Linear Regression suffers from overfitting and can't deal with collinear data. When there are many features in the dataset and even some of them are not relevant to the predictive model. This makes the model more complex with a too-inaccurate prediction on the test set (or overfitting). Such a model with high variance does not generalize on the new data. So, to deal with these issues, we include both L-2 and L-1 norm regularization to get the benefits of both Ridge and Lasso at the same time. The resultant model has better predictive power than Lasso. It performs feature selection and also makes the hypothesis simpler. The modified cost function for **Elastic-Net Regression** is given below:

$$\frac{1}{m}\left[\sum_{l=1}^{m}\left(y^{(i)}-h\left(x^{(i)}\right)\right)^{2}+\lambda_{1}\sum_{j=1}^{n}w_{j}+\lambda_{2}\sum_{j=1}^{n}w_{j}^{2}\right]$$

where,

- **w(j)** represents the weight for the j feature.

  th

- **n** is the number of features in the dataset.
- l**ambda1** is the regularization strength for the L1 norm.
- **lambda2** is the regularization strength for the L2 norm.

Introduction:

Elastic-Net Regression is a modification of Linear Regression which shares the same hypothetical function for prediction. The cost function of Linear Regression is represented by *J*.

$$\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}-h\left(x^{(i)}\right)\right)^{2}$$

```
Here, m is the total number of training examples in the dat
aset.
h(x(i)) represents the hypothetical function for predictio
n.
y(i) represents the value of target variable for ith traini
ng example.
```

Linear Regression suffers from overfitting and can’t deal with collinear data. When there are many features in the dataset and even some of them are not relevant for the predictive model. This makes the model more complex with a too inaccurate prediction on the test set (or overfitting). Such a model with high variance does not generalize on the new data. So, to deal with these issues, we include both L-2 and L-1 norm regularization to get the benefits of both Ridge and Lasso at the same time. The resultant model has better predictive power than Lasso. It performs feature selection and also makes the hypothesis simpler. The modified cost function for Elastic-Net Regression is given below :

$$\frac{1}{m}\left[\sum_{l=1}^{m}\left(y^{(i)}-h\left(x^{(i)}\right)\right)^{2}+\lambda_{1}\sum_{j=1}^{n}w_{j}+\lambda_{2}\sum_{j=1}^{n}w_{j}^{2}\right]$$

```
Here,w(j) represents the weight for jth feature.
n is the number of features in the dataset.
lambda1 is the regularization strength for L-1 norm.
lambda2 is the regularization strength for L-2 norm.
```

Mathematical Intuition:

During gradient descent optimization of its cost function, added L-2 penalty term leads to reduces the weights of the model close to zero. Due to the penalization of weights, the hypothesis gets simpler, more generalized, and less prone to overfitting. Added L1 penalty shrunk weights close to zero or zero. Those weights which are shrunken to zero eliminates the features present in the hypothetical function. Due to this, irrelevant features don't participate in the predictive model. This penalization of weights makes the hypothesis more predictive which encourages the sparsity (model with few parameters).

Different cases for tuning values of lambda1 and lamda2.

1. If lambda1 and lambda2 are set to be 0, Elastic-Net Regression equals Linear Regression.
2. If lambda1 is set to be 0, Elastic-Net Regression equals Ridge Regression.
3. If lambda2 is set to be 0, Elastic-Net Regression equals Lasso Regression.
4. If lambda1 and lambda2 are set to be infinity, all weights are shrunk to zero

So, we should set lambda1 and lambda2 somewhere in between 0 and infinity.

Implementation:

Dataset used in this implementation can be downloaded from the **link**.

It has 2 columns — “*YearsExperience*” and “*Salary*” for 30 employees in a company. So in this, we will train an Elastic-Net Regression model to learn the correlation between the number of years of experience of each employee and their respective salary. Once the model is trained, we will be able to predict the salary of an employee on the basis of his years of experience.

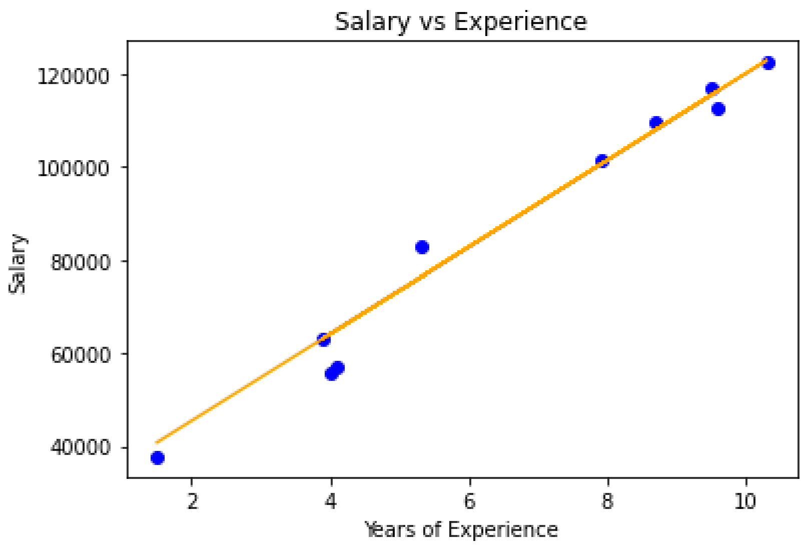
Code:

```
# Importing libraries

import numpy as np

import pandas as pd

from sklearn.model_selection import train_test_split

import matplotlib.pyplot as plt

# Elastic Net Regression

class ElasticRegression() :

    def __init__( self, learning_rate, iterations, l1_penal
ity, l2_penality ) :

        self.learning_rate = learning_rate

        self.iterations = iterations

        self.l1_penality = l1_penality

        self.l2_penality = l2_penality

    # Function for model training

    def fit( self, X, Y ) :

        # no_of_training_examples, no_of_features
        self.m, self.n = X.shape

        # weight initialization

        self.W = np.zeros( self.n )

        self.b = 0

        self.X = X

        self.Y = Y

        # gradient descent learning

        for i in range( self.iterations ) :

            self.update_weights()

        return self

    # Helper function to update weights in gradient descent

    def update_weights( self ) :

        Y_pred = self.predict( self.X )

        # calculate gradients

        dW = np.zeros( self.n )

        for j in range( self.n ) :

            if self.W[j] > 0 :

                dW[j] = ( - ( 2 * ( self.X[:,j] ).dot( sel
f.Y - Y_pred ) ) +

                        self.l1_penality + 2 * self.l2_pena
lity * self.W[j] ) / self.m

            else :

                dW[j] = ( - ( 2 * ( self.X[:,j] ).dot( sel
f.Y - Y_pred ) )

                        - self.l1_penality + 2 * self.l2_pe
nality * self.W[j] ) / self.m


        db = - 2 * np.sum( self.Y - Y_pred ) / self.m

        # update weights

        self.W = self.W - self.learning_rate * dW

        self.b = self.b - self.learning_rate * db

        return self

    # Hypothetical function h( x )

    def predict( self, X ) :

        return X.dot( self.W ) + self.b

# Driver Code

def main() :

    # Importing dataset

    df = pd.read_csv( "salary_data.csv" )

    X = df.iloc[:,:-1].values

    Y = df.iloc[:,1].values
    # Splitting dataset into train and test set

    X_train, X_test, Y_train, Y_test = train_test_split( X, 
Y,

                                            test_size = 1/
3, random_state = 0 )

    # Model training

    model = ElasticRegression( iterations = 1000,

                    learning_rate = 0.01, l1_penality = 50
0, l2_penality = 1 )

    model.fit( X_train, Y_train )

    # Prediction on test set

    Y_pred = model.predict( X_test )

    print( "Predicted values ", np.round( Y_pred[:3], 2 ) )

    print( "Real values  ", Y_test[:3] )

    print( "Trained W    ", round( model.W[0], 2 ) )

    print( "Trained b    ", round( model.b, 2 ) )

    # Visualization on test set

    plt.scatter( X_test, Y_test, color = 'blue' )

    plt.plot( X_test, Y_pred, color = 'orange' )

    plt.title( 'Salary vs Experience' )
    plt.xlabel( 'Years of Experience' )

    plt.ylabel( 'Salary' )

    plt.show()


if __name__ == "__main__" :

    main()
```

Output:

```
Predicted values  [ 40837.61 122887.43  65079.6 ]
Real values       [ 37731 122391  57081]
Trained W         9323.84
Trained b         26851.84
```

Elastic-Net Model Visualization



Note: Elastic-Net Regression automates certain parts of model selection and leads to dimensionality reduction which makes it a computationally efficient model.

Bayesian Linear Regression

As the name suggests this algorithm is purely based on **Bayes Theorem**. Because of this reason only we do not use the Least Square method to determine the coefficients of the regression model. So, the technique which is used here to find the model weights and parameters relies on features posterior distribution and this provides an extra stability factor to the regression model which is based on this technique.

```
class LinearRegression:
    def __init__(self):
        self.parameters = {}def forward_propagation(self, t
rain_input):
    m = self.parameters['m']
    c = self.parameters['c']
    predictions = np.multiply(m, train_input) + c
    return predictions

def cost_function(self, predictions, train_output):
    cost = np.mean((train_output - predictions) ** 2)
    return cost

def backward_propagation(self, train_input, train_output, p
redictions):
    derivatives = {}
    df = (train_output - predictions) * -1
    dm = np.mean(np.multiply(train_input, df))
    dc = np.mean(df)
    derivatives['dm'] = dm
    derivatives['dc'] = dc
    return derivatives

def update_parameters(self, derivatives, learning_rate):
    self.parameters['m'] = self.parameters['m'] - learning_
rate * derivatives['dm']
    self.parameters['c'] = self.parameters['c'] - learning_
rate * derivatives['dc']

def train(self, train_input, train_output, learning_rate, i
ters):
    #initialize random parameters
    self.parameters['m'] = np.random.uniform(0,1) * -1
    self.parameters['c'] = np.random.uniform(0,1) * -1

    #initialize loss
    self.loss = []

    #iterate
    for i in range(iters):
        #forward propagation
        predictions = self.forward_propagation(train_input)

        #cost function
        cost = self.cost_function(predictions, train_outpu
t)

        #append loss and print
        self.loss.append(cost)
        print("Iteration = {}, Loss = {}".format(i+1, cos
t))

        #back propagation
        derivatives = self.backward_propagation(train_inpu
t, train_output, predictions)

        #update parameters
        self.update_parameters(derivatives, learning_rate)

    return self.parameters, self.loss
```

Linear regression is a popular regression approach in machine learning. Linear regression is based on the assumption that the underlying data is normally distributed and that all relevant predictor variables have a linear relationship with the outcome. But In the real world, this is not always possible, it will follows these assumptions, Bayesian regression could be the better choice.

Bayesian regression employs prior belief or knowledge about the data to “learn” more about it and create more accurate predictions. It also takes into account the data’s uncertainty and leverages prior knowledge to provide more precise estimates of the data. As a result, it is an ideal choice when the data is complex or ambiguous.

Bayesian regression uses a Bayes algorithm to estimate the parameters of a linear regression model from data, including prior knowledge about the parameters. Because of its probabilistic character, it can produce more accurate estimates for regression parameters than ordinary least squares (OLS) linear regression, provide a measure of uncertainty in the estimation, and make stronger conclusions than OLS. Bayesian regression can also be utilized for related regression analysis tasks like model selection and outlier detection.

Bayesian Regression

Bayesian regression is a type of linear regression that uses Bayesian statistics to estimate the unknown parameters of a model. It uses Bayes’ theorem to estimate the likelihood of a set of parameters given observed data. The goal of Bayesian regression is to find the best estimate of the parameters of a linear model that describes the relationship between the independent and the dependent variables.

The main difference between traditional linear regression and Bayesian regression is the underlying assumption regarding the data-generating process. Traditional linear regression assumes that data follows a Gaussian or normal distribution, while Bayesian regression has stronger assumptions about the nature of the data and puts a prior probability distribution on the parameters. Bayesian regression also enables more flexibility as it allows for additional parameters or prior distributions, and can be used to construct an arbitrarily complex model that explicitly expresses prior beliefs about the data. Additionally, Bayesian regression provides more accurate predictive measures from fewer data points and is able to construct estimates for uncertainty around the estimates. On the other hand, traditional linear regressions are easier to implement and generally faster with simpler models and can provide good results when the assumptions about the data are valid.

Bayesian Regression can be very useful when we have insufficient data in the dataset or the data is poorly distributed. The output of a Bayesian Regression model is obtained from a probability distribution, as compared to regular regression techniques where the output is just obtained from a single value of each attribute.

Some Dependent Concepts for Bayesian Regression

The important concepts in Bayesian Regression are as follows:

Bayes Theorem

Bayes Theorem gives the relationship between an event's prior probability and its posterior probability after evidence is taken into account. It states that the conditional probability of an event is equal to the probability of the event given certain conditions multiplied by the prior probability of the event, divided by the probability of the conditions.

i.e

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

.

Where P(A|B) is the probability of event A occurring given that event B has already occurred, P(B|A) is the probability of event B occurring given that event A has already occurred, P(A) is the probability of event A occurring and P(B) is the probability of event B occurring.

Maximum Likelihood Estimation (MLE)

MLE is a method used to estimate the parameters of a statistical model by maximizing the likelihood function. it seeks to find the parameter values that make the observed data most probable under the assumed model. MLE does not incorporate any prior information or assumptions about the parameters, and it provides point estimates of the parameters

Maximum A Posteriori (MAP) Estimation

MAP estimation is a Bayesian approach that combines prior information with the likelihood function to estimate the parameters. It involves finding the parameter values that maximize the posterior distribution, which is obtained by applying Bayes' theorem. In MAP estimation, a prior distribution is specified for the parameters, representing prior beliefs or knowledge about their values. The likelihood function is then multiplied by the prior distribution to obtain the joint distribution, and the parameter values that maximize this joint distribution are selected as the MAP estimates. MAP estimation provides point estimates of the parameters, similar to MLE, but incorporates prior information.

Need for Bayesian Regression

There are several reasons why Bayesian regression is useful over other regression techniques. Some of them are as follows:

1. Bayesian regression also uses the prior belief about the parameters in the analysis. which makes it useful when there is limited data available and the prior knowledge are relevant. By combining prior knowledge with the observed data, Bayesian regression provides more informed and potentially more accurate estimates of the regression parameters.
2. Bayesian regression provides a natural way to measure the uncertainty in the estimation of regression parameters by generating the posterior distribution, which captures the uncertainty in the parameter values, as opposed to the single point estimate that is produced by standard regression techniques. This distribution offers a range of acceptable values for the parameters and can be used to compute trustworthy intervals or Bayesian confidence intervals.
3. In order to incorporate complicated correlations and non-linearities, Bayesian regression provides flexibility by offering a framework for integrating various prior distributions, which makes it capable to handle situations where the basic assumptions of standard regression techniques, like linearity or homoscedasticity, may not be true. It enables the modeling of more realistic and nuanced relationships between the predictors and the response variable.
4. Bayesian regression facilitates model selection and comparison by calculating the posterior probabilities of different models.
5. Bayesian regression can handle outliers and influential observations more effectively compared to classical regression methods. It provides a more robust approach to regression analysis, as extreme or influential observations have a lesser impact on the estimation.

Implementation of Bayesian Regression

Let’s independent features for linear regression is

$$X = \{x_1, x_2, ..., x_P\}$$

, where x_i represents the i

th

independent features and target variables will be Y. Assume we have n samples of (X, y)

The linear relationship between the dependent variable Y and the independent features X can be represented as:

$$y = w_{+w_{x_{+w_{x_{+\ldots+w_{x_{+\epsilon}}}}}}}$$

or

$$y = f(x, w) + \epsilon$$

Here,

$$w = \{w_{,w_{,w_{,\ldots,w_{\}}}}}$$

are the regression coefficients, representing the relationship between the independent variables and the dependent variable, and ε is the error term.We assume that the errors (ε) follow a normal distribution with mean 0 and constant variance

$$\sigma^2$$

i.e.

$$(\epsilon \sim N(0, \sigma^2))$$

. This assumption allows us to model the distribution of the target variable around the predicted values.

Likelihood Function

The probability distribution that builds the relationship between the independent features and the regression coefficients is known as likehood. It describes the probability of obtaining a certain result given some combination of regression coefficients.

Assumptions:

- The errors are independent and identical and follow a normal distribution with mean 0 and variance σ².

$$\epsilon = \{\epsilon_1, \epsilon_{,\ldots,\epsilon\}}$$

This means the target variable Y, given the predictor X

1

, X

2

, ..., X

p

follows a normal distribution with mean

$$\mu = f(x, w) = w_0 + w_1x_1 + w_2x_2 + \ldots + w_px_p$$

and variance

$$\sigma^2$$

.

Therefore, The conditional probability density function (PDF) of Y given the predictor variables will be:

$$\begin{aligned} P(y|x, w, \sigma^2) &= N(f(x, w), \sigma^2) \\ &= \frac{1}{\sqrt{2\pi\sigma^2}} e^{\left[-\frac{(y-f(x,w))^2}{2\sigma^2}\right]} \\ &= \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[\frac{y - (w_0 + w_1x_1 + w_2x_2 + \cdots + w_Px_P)}{2\sigma^2}\right] \end{aligned}$$

The likelihood function, for the n observations having each observation

$$(x_{i1}, x_{i2}, \cdots, x_{iP}, y)$$

that follows a normal distribution with mean

$$\mu_i = w_0 + w_1 x_{i1} + w_2 x_{i2} + \cdots + w_P x_{iP}$$

and variance

$$\sigma^2$$

, will be the joint probability density function (PDF) of the dependent variables, and can be written as the product of the individual PDFs::

$$\begin{aligned} L(Y|X, w, \sigma^2) &= P(y_1|x_{11}, x_{12}, \cdots, x_{1P}) \cdot P(y_2|x_{21}, x_{22}, \cdots, x_{2P}) \cdots P(y_n|x_{n1}, x_{n2}, \cdots, x_{nP}) \\ &= N(f(x_{1p}, w), \sigma^2) \cdot N(f(x_{2p}, w), \sigma^2) \cdots N(f(x_{np}, w), \sigma^2) \cdot \\ &= \prod_{i=1}^{N} [N(f(x_{ip}, w), \sigma^2)] \end{aligned}$$

To simplify calculations, we can take the logarithm of the likelihood function:

$$\begin{aligned} \ln(L(Y|X, w, \sigma^2)) &= \ln\left[\prod_{i=1}^{N} [N(f(x_{ip}, w), \sigma^2)]\right] \\ &= \ln[N(f(x_{1p}, w), \sigma^2)] + \ln[N(f(x_{2p}, w), \sigma^2)] + \cdots + \ln[N(f(x_{np}, w), \sigma^2)] \\ &= \sum_{i=1}^{N} \ln\left[\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - (w_0 + w_1 x_{i1} + w_2 x_{i2} + \cdots + w_P x_{iP}))^2}{2\sigma^2}\right)\right] \end{aligned}$$

Precision:

\beta = \frac{1}{\sigma^2}

Using the conditional PDF expression from the previous response, we substitute it into the likelihood function:

$$\ln(L(y|x, w, \sigma^2)) = \frac{N}{2}\ln(2\pi) - \frac{N}{2}\ln(\beta) - \frac{\beta}{2}\sum_{i=1}^{N}\left[(y - f(x_i, w))^2\right]$$

Negative Log likelihood

$$-\ln(L(y|x, w, \sigma^2)) = \frac{\beta}{2}\sum_{i=1}^{N}\left[(y - f(x_i, w))^2\right] - \frac{N}{2}\ln(2\pi) + \frac{N}{2}\ln(\beta)$$

Here,

$$\ln(2\pi)$$

and

$$\ln(\beta)$$

are constant, So,

$$-\ln(L(y|x,w,\sigma^2)) = \frac{\beta}{2}\sum_{i=1}^{N}\left[(y-f(x_i,w))^2\right] + \text{constant}$$

Prior:

Prior is the initial belief or probability about the parameter before observing the data. It is the information or assumption about the parameters.

In Maximum A Posteriori (MAP) estimation, we incorporate prior knowledge or beliefs about the parameters into the estimation process. We represent this prior information using a prior distribution, denoted by P(w|\alpha) =N(0,\alpha^{-1}I)

Posterior Distribution:

The posterior distribution is the updated beliefs or probability distribution of parameters, which comes after taking into account the prior distribution of parameters and the observed data.

Using Bayes' theorem, we can express the posterior distribution in terms of the likelihood function and the prior distribution:

$$P(w|X,\alpha,\beta^{-1}) = \frac{L(Y|X,w,\beta^{-1})\cdot P(w|\alpha)}{P(Y|X)}$$

P(Y|X) is the marginal probability of the observed data, which acts as a normalizing constant. Since it does not depend on the parameter values, we can ignore it in the optimization process.

$$P(w|X,\alpha,\beta^{-1}) \propto (L(Y|X,w,\beta^{-1})\cdot P(w|\alpha))$$

In the above formula,

- Log-likelihood : is normal distribution

$$L(Y|X,w,\beta^{-1})$$

- Prior : is uniform

$$P(w|\alpha)$$

- So, the Posterior will also be the normal distribution.

In practice, it is often more convenient to work with the log of the posterior distribution, known as the log-posterior:

For getting maximum posterior distribution, we use the negative likelihood

$$\begin{aligned}\hat{w} &= \ln[(L(Y|X,w,\beta^{-1})\cdot P(w|\alpha))] \\ &= \ln[L(Y|X,w,\beta^{-1})] + \ln[P(w|\alpha)] \\ &= \frac{\beta}{2}\sum_{i=1}^{N}(y_i - f(x,w)) + \frac{\alpha}{2}w^Tw\end{aligned}$$

Maximum of Posterior is given by a minimum of

$$\tfrac{\beta}{2}\textstyle\sum_{i=1}^{N}(y_i - f(x,w)) + \tfrac{\alpha}{2}w^Tw$$

Implementation of Bayesian Regression Using Python:

```
#Import the necessary libraries
import torch
import pyro
import pyro.distributions as dist
from pyro.infer import SVI, Trace_ELBO, Predictive
from pyro.optim import Adam
import matplotlib.pyplot as plt
import seaborn as sns


# Generate some sample data
torch.manual_seed(0)
X = torch.linspace(0, 10, 100)
true_slope = 2
```

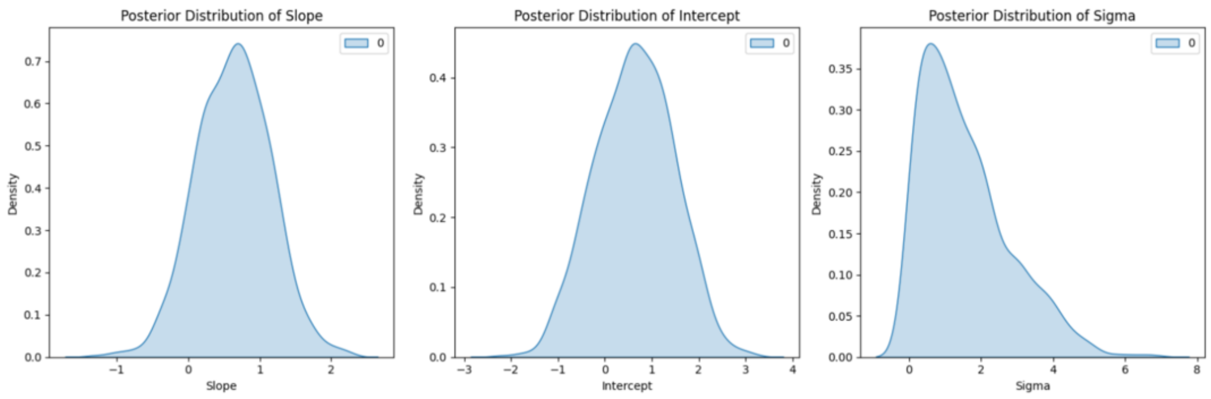
```
true_intercept = 1
Y = true_intercept + true_slope * X + torch.randn(100)

# Define the Bayesian regression model
def model(X, Y):
    # Priors for the parameters
    slope = pyro.sample("slope", dist.Normal(0, 10))
    intercept = pyro.sample("intercept", dist.Normal(0, 1
0))
    sigma = pyro.sample("sigma", dist.HalfNormal(1))

    # Expected value of the outcome
    mu = intercept + slope * X

    # Likelihood (sampling distribution) of the observation
s
    with pyro.plate("data", len(X)):
        pyro.sample("obs", dist.Normal(mu, sigma), obs=Y)

# Run Bayesian inference using SVI (Stochastic Variational
Inference)
def guide(X, Y):
    # Approximate posterior distributions for the parameter
s
    slope_loc = pyro.param("slope_loc", torch.tensor(0.0))
    slope_scale = pyro.param("slope_scale", torch.tensor(1.
0),
                             constraint=dist.constraints.pos
itive)
    intercept_loc = pyro.param("intercept_loc", torch.tenso
r(0.0))
    intercept_scale = pyro.param("intercept_scale", torch.t
ensor(1.0),
                                 constraint=dist.constraint
s.positive)
    sigma_loc = pyro.param("sigma_loc", torch.tensor(1.0),
                         constraint=dist.constraints.positiv
e)
```

```
    # Sample from the approximate posterior distributions
    slope = pyro.sample("slope", dist.Normal(slope_loc, slo
pe_scale))
    intercept = pyro.sample("intercept", dist.Normal(interc
ept_loc,
                                                     interce
pt_scale))
    sigma = pyro.sample("sigma", dist.HalfNormal(sigma_lo
c))

# Initialize the SVI and optimizer
optim = Adam({"lr": 0.01})
svi = SVI(model, guide, optim, loss=Trace_ELBO())

# Run the inference loop
num_iterations = 1000
for i in range(num_iterations):
    loss = svi.step(X, Y)
    if (i + 1) % 100 == 0:
        print(f"Iteration {i + 1}/{num_iterations} - Loss:
{loss}")

# Obtain posterior samples using Predictive
predictive = Predictive(model, guide=guide, num_samples=100
0)
posterior = predictive(X, Y)

# Extract the parameter samples
slope_samples = posterior["slope"]
intercept_samples = posterior["intercept"]
sigma_samples = posterior["sigma"]

# Compute the posterior means
slope_mean = slope_samples.mean()
intercept_mean = intercept_samples.mean()
sigma_mean = sigma_samples.mean()
```

```
# Print the estimated parameters
print("Estimated Slope:", slope_mean.item())
print("Estimated Intercept:", intercept_mean.item())
print("Estimated Sigma:", sigma_mean.item())



# Create subplots
fig, axs = plt.subplots(1, 3, figsize=(15, 5))

# Plot the posterior distribution of the slope
sns.kdeplot(slope_samples, shade=True, ax=axs[0])
axs[0].set_title("Posterior Distribution of Slope")
axs[0].set_xlabel("Slope")
axs[0].set_ylabel("Density")

# Plot the posterior distribution of the intercept
sns.kdeplot(intercept_samples, shade=True, ax=axs[1])
axs[1].set_title("Posterior Distribution of Intercept")
axs[1].set_xlabel("Intercept")
axs[1].set_ylabel("Density")

# Plot the posterior distribution of sigma
sns.kdeplot(sigma_samples, shade=True, ax=axs[2])
axs[2].set_title("Posterior Distribution of Sigma")
axs[2].set_xlabel("Sigma")
axs[2].set_ylabel("Density")

# Adjust the layout
plt.tight_layout()

# Show the plot
plt.show()
```

Output:

```
Iteration 100/1000 - Loss: 26402.827576458454
Iteration 200/1000 - Loss: 868.8797441720963
Iteration 300/1000 - Loss: 375.8896267414093
Iteration 400/1000 - Loss: 410.9912616610527
Iteration 500/1000 - Loss: 1326.6764334440231
Iteration 600/1000 - Loss: 6281.599338412285
Iteration 700/1000 - Loss: 928.897826552391
Iteration 800/1000 - Loss: 162330.28921669722
Iteration 900/1000 - Loss: 803.6285791993141
Iteration 1000/1000 - Loss: 16296.759144246578
Estimated Slope: 0.6329146027565002
Estimated Intercept: 0.632161021232605
Estimated Sigma: 1.5464321374893188
```



Advantages of Bayesian Regression:

- Very effective when the size of the dataset is small.
- Particularly well-suited for on-line based learning (data is received in real-time), as compared to batch-based learning, where we have the entire dataset on our hands before we start training the model. This is because Bayesian Regression doesn't need to store data.
- The Bayesian approach is a tried and tested approach and is very robust, mathematically. So, one can use this without having any extra prior knowledge about the dataset.
- Bayesian regression methods employ skewed distributions that let you include outside information in your model.

Disadvantages of Bayesian Regression:

- The inference of the model can be time-consuming.

- If there is a large amount of data available for our dataset, the Bayesian approach is not worth it and the regular frequentist approach does a more efficient job
- If your code is in a setting where installing new packages is challenging and there are strong dependency controls, this might be an issue.
- Many of the basic mistakes that commonly plague traditional frequentist regression models still apply to Bayesian models. For instance, Bayesian models still depend on the linearity of the relationship between the characteristics and the outcome variable.

When to use Bayesian Regression:

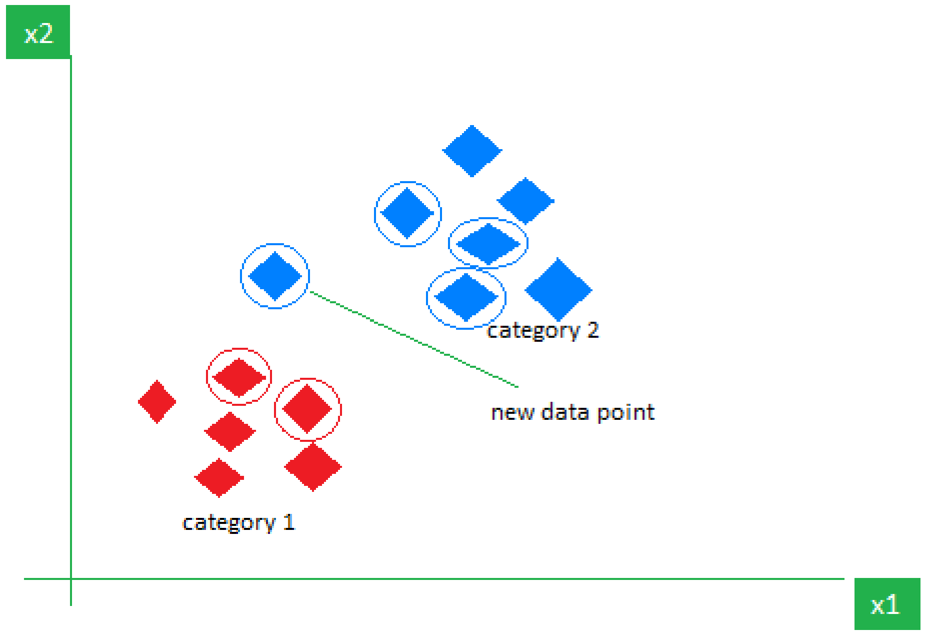
- **Small sample size:** When you have a tiny sample size, Bayesian inference is frequently quite helpful. Bayesian regression is a wonderful choice if you need to develop a very complicated model but do not have access to a lot of data. In fact, it's probably the greatest choice you have! Large sample sizes are necessary for many other machine-learning models to work properly.
- **Strong prior knowledge:** The easiest method to include strong external knowledge into your model is by utilising a Bayesian model. The influence of the prior knowledge will be more obvious the smaller the dataset size you work with.

Conclusion:

So now that you know how Bayesian regressors work and when to use them, you should try using it next time you want to perform a regression task, especially if the dataset is small.

k-nearest neighbour algorithm

In this article, we will learn about a supervised learning algorithm that is popularly known as the KNN or the k – Nearest Neighbours.



Now, given another set of data points (also called testing data), allocate these points to a group by analyzing the training set. Note that the unclassified points are marked as ‘White’.

Intuition Behind KNN Algorithm

If we plot these points on a graph, we may be able to locate some clusters or groups. Now, given an unclassified point, we can assign it to a group by observing what group its nearest neighbors belong to. This means a point close to a cluster of points classified as ‘Red’ has a higher probability of getting classified as ‘Red’.

Intuitively, we can see that the first point (2.5, 7) should be classified as ‘Green’ and the second point (5.5, 4.5) should be classified as ‘Red’.

Distance Metrics Used in KNN Algorithm

As we know that the KNN algorithm helps us identify the nearest points or the groups for a query point. But to determine the closest groups or the nearest points for a query point we need some metric. For this purpose, we use below distance metrics:

- **Euclidean Distance**
- **Manhattan Distance**
- **Minkowski Distance**

Euclidean Distance

This is nothing but the cartesian distance between the two points which are in the plane/hyperplane. Euclidean distance can also be visualized as the length of the straight line that joins the two points which are into consideration. This metric helps us calculate the net displacement done between the two states of an object.

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

Manhattan Distance

This distance metric is generally used when we are interested in the total distance traveled by the object instead of the displacement. This metric is calculated by summing the absolute difference between the coordinates of the points in n-dimensions.

$$d\left(x,y\right)=\sum_{i=1}^{n}\left|x_i-y_i\right|$$

Minkowski Distance

We can say that the Euclidean, as well as the Manhattan distance, are special cases of the Minkowski distance.

$$d\left(x,y\right)=\left(\sum_{i=1}^{n}\left(x_i-y_i\right)^p\right)^{\frac{1}{p}}$$

From the formula above we can say that when p = 2 then it is the same as the formula for the Euclidean distance and when p = 1 then we obtain the formula for the Manhattan distance.

The above-discussed metrics are most common while dealing with a **Machine Learning** problem but there are other distance metrics as well like **Hamming**

Distance which come in handy while dealing with problems that require overlapping comparisons between two vectors whose contents can be boolean as well as string values.

How to choose the value of k for KNN Algorithm?

The value of k is very crucial in the KNN algorithm to define the number of neighbors in the algorithm. The value of k in the k-nearest neighbors (k-NN) algorithm should be chosen based on the input data. If the input data has more outliers or noise, a higher value of k would be better. It is recommended to choose an odd value for k to avoid ties in classification. **Cross-validation** methods can help in selecting the best k value for the given dataset.

Applications of the KNN Algorithm

- **Data Preprocessing** – While dealing with any Machine Learning problem we first perform the **EDA** part in which if we find that the data contains missing values then there are multiple imputation methods are available as well. One of such method is **KNN Imputer** which is quite effective ad generally used for sophisticated imputation methodologies.
- **Pattern Recognition** – KNN algorithms work very well if you have trained a KNN algorithm using the MNIST dataset and then performed the evaluation process then you must have come across the fact that the accuracy is too high.
- **Recommendation Engines** – The main task which is performed by a KNN algorithm is to assign a new query point to a pre-existed group that has been created using a huge corpus of datasets. This is exactly what is required in the **recommender systems** to assign each user to a particular group and then provide them recommendations based on that group's preferences.

Advantages of the KNN Algorithm

- **Easy to implement** as the complexity of the algorithm is not that high.

- **Adapts Easily** – As per the working of the KNN algorithm it stores all the data in memory storage and hence whenever a new example or data point is added then the algorithm adjusts itself as per that new example and has its contribution to the future predictions as well.
- **Few Hyperparameters** – The only parameters which are required in the training of a KNN algorithm are the value of k and the choice of the distance metric which we would like to choose from our evaluation metric.

Disadvantages of the KNN Algorithm

- **Does not scale** – As we have heard about this that the KNN algorithm is also considered a Lazy Algorithm. The main significance of this term is that this takes lots of computing power as well as data storage. This makes this algorithm both time-consuming and resource exhausting.
- **Curse of Dimensionality** – There is a term known as the peaking phenomenon according to this the KNN algorithm is affected by the **curse of dimensionality** which implies the algorithm faces a hard time classifying the data points properly when the dimensionality is too high.
- **Prone to Overfitting** – As the algorithm is affected due to the curse of dimensionality it is prone to the problem of overfitting as well. Hence generally **feature selection** as well as **dimensionality reduction**techniques are applied to deal with this problem.

Example Program:

Assume 0 and 1 as the two classifiers (groups).

```
# Python3 program to find groups of unknown
# Points using K nearest neighbour algorithm.

import math

def classifyAPoint(points,p,k=3):
    '''
    This function finds the classification of p using
    k nearest neighbor algorithm. It assumes only two
    groups and returns 0 if p belongs to group 0, else
    1 (belongs to group 1).
    Parameters -
        points: Dictionary of training points having two ke
ys - 0 and 1
                Each key have a list of training data point
s belong to that

        p : A tuple, test data point of the form (x,y)

        k : number of nearest neighbour to consider, defaul
t is 3
    '''

    distance=[]
    for group in points:
        for feature in points[group]:

            #calculate the euclidean distance of p from tra
ining points
            euclidean_distance = math.sqrt((feature[0]-p
[0])**2 +(feature[1]-p[1])**2)

            # Add a tuple of form (distance,group) in the d
istance list
            distance.append((euclidean_distance,group))

    # sort the distance list in ascending order
    # and select first k distances
    distance = sorted(distance)[:k]

    freq1 = 0 #frequency of group 0
    freq2 = 0 #frequency og group 1

    for d in distance:
        if d[1] == 0:
            freq1 += 1
        else if d[1] == 1:
            freq2 += 1
    return 0 if freq1>freq2 else 1

# driver function
def main():

    # Dictionary of training points having two keys - 0 and 
1
    # key 0 have points belong to class 0
    # key 1 have points belong to class 1

    points = {0:[(1,12),(2,5),(3,6),(3,10),(3.5,8),(2,11),
(2,9),(1,7)],
                1:[(5,3),(3,2),(1.5,9),(7,2),(6,1),(3.8,1),(5.
6,4),(4,2),(2,5)]}

    # testing point p(x,y)
    p = (2.5,7)

    # Number of neighbours
    k = 3

    print("The value classified to unknown point is: {}".\
        format(classifyAPoint(points,p,k)))

if __name__ == '__main__':
    main()

# This code is contributed by Atul Kumar (www.fb.com/atul.k
r.007)
```

Output:

```
The value classified as an unknown point is 0.
```

Time Complexity: O(N * logN)

Auxiliary Space: O(1)

What is K-Nearest Neighbors Algorithm?

K-Nearest Neighbours is one of the most basic yet essential classification algorithms in Machine Learning. It belongs to the **supervised learning** domain and finds intense application in pattern recognition, **data mining**, and intrusion detection.

It is widely disposable in real-life scenarios since it is non-parametric, meaning, it does not make any underlying assumptions about the distribution of data (as opposed to other algorithms such as GMM, which assume a **Gaussian distribution** of the given data). We are given some prior data (also called training data), which classifies coordinates into groups identified by an attribute.

As an example, consider the following table of data points containing two features:

Supervised Learning :

It is the learning where the value or result that we want to predict is within the training data (labeled data) and the value which is in data that we want to study is known as Target or Dependent Variable or *Response Variable*.

All the other columns in the dataset are known as the Feature or Predictor Variable or Independent Variable.

Supervised Learning is classified into two categories:

1. **Classification**: Here our target variable consists of the categories.
2. **Regression**: Here our target variable is continuous and we usually try to find out the line of the curve.

As we have understood that to carry out supervised learning we need labeled data. How we can get labeled data? There are various ways to get labeled data:

1. Historical labeled Data
2. Experiment to get data: We can perform experiments to generate labeled data like A/B Testing.
3. Crowd-sourcing

Now it’s time to understand algorithms that can be used to solve supervised machine learning problem. In this post, we will be using popular **scikit-**

learn package.

> Note: There are few other packages as well like TensorFlow, Keras etc to perform supervised learning.

This algorithm is used to solve the classification model problems. K-nearest neighbor or K-NN algorithm basically creates an imaginary boundary to classify the data. When new data points come in, the algorithm will try to predict that to the nearest of the boundary line.

Therefore, larger k value means smother curves of separation resulting in less complex models. Whereas, smaller k value tends to overfit the data and resulting in complex models.

Note: It’s very important to have the right k-value when analyzing the dataset to avoid overfitting and underfitting of the dataset.

Using the k-nearest neighbor algorithm we fit the historical data (or train the model) and predict the future.

Example of the k-nearest neighbor algorithm

```
# Import necessary modules
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris

# Loading data
irisData = load_iris()

# Create feature and target arrays
X = irisData.data
y = irisData.target

# Split into training and test set
X_train, X_test, y_train, y_test = train_test_split(
            X, y, test_size = 0.2, random_state=42)

knn = KNeighborsClassifier(n_neighbors=7)
```

```
knn.fit(X_train, y_train)

# Predict on dataset which model has not seen before
print(knn.predict(X_test))
```

In the example shown above following steps are performed:

1. The k-nearest neighbor algorithm is imported from the scikit-learn package.
2. Create feature and target variables.
3. Split data into training and test data.
4. Generate a k-NN model using neighbors value.
5. Train or fit the data into the model.
6. Predict the future.

We have seen how we can use K-NN algorithm to solve the supervised machine learning problem. But how to measure the accuracy of the model?

Consider an example shown below where we predicted the performance of the above model:

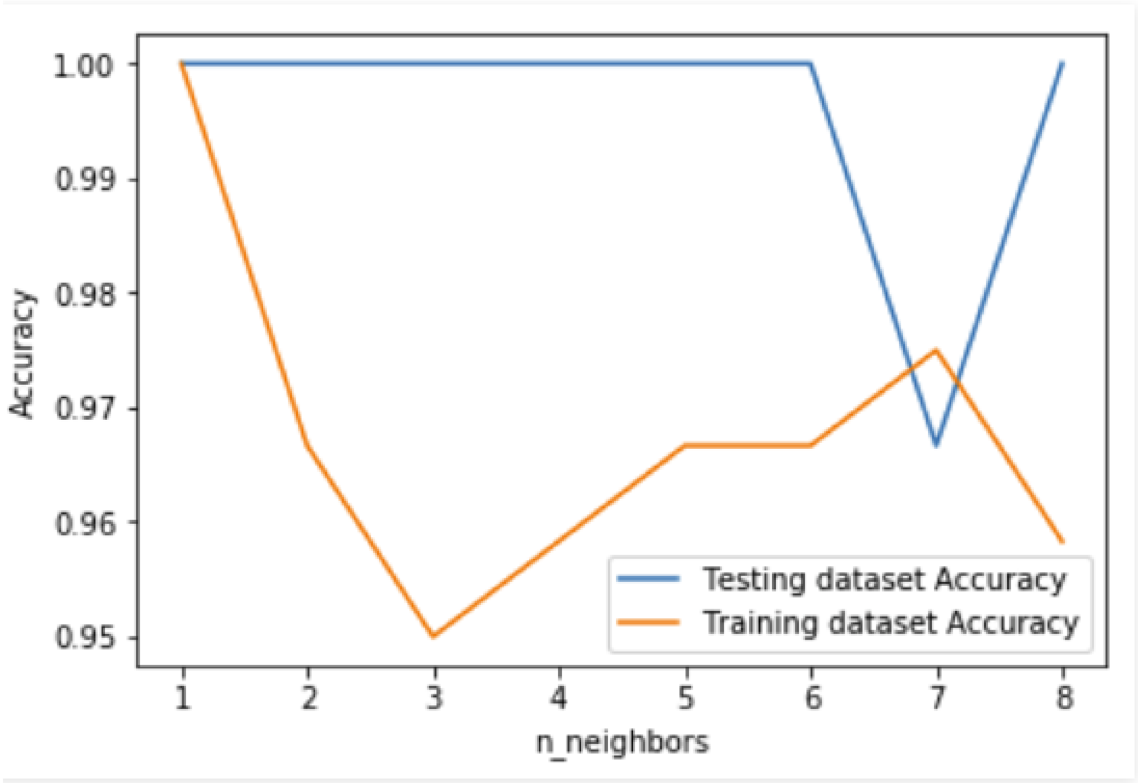
```
# Import necessary modules
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris

# Loading data
irisData = load_iris()

# Create feature and target arrays
X = irisData.data
y = irisData.target

# Split into training and test set
X_train, X_test, y_train, y_test = train_test_split(
            X, y, test_size = 0.2, random_state=42)

knn = KNeighborsClassifier(n_neighbors=7)
```

```
knn.fit(X_train, y_train)

# Calculate the accuracy of the model
print(knn.score(X_test, y_test))
```

Model Accuracy:

So far so good. But how to decide the right k-value for the dataset? Obviously, we need to be familiar to data to get the range of expected k-value, but to get the exact k-value we need to test the model for each and every expected k-value. Refer to the example shown below.

```
# Import necessary modules
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
import numpy as np
import matplotlib.pyplot as plt

irisData = load_iris()

# Create feature and target arrays
X = irisData.data
y = irisData.target

# Split into training and test set
X_train, X_test, y_train, y_test = train_test_split(
            X, y, test_size = 0.2, random_state=42)

neighbors = np.arange(1, 9)
train_accuracy = np.empty(len(neighbors))
test_accuracy = np.empty(len(neighbors))

# Loop over K values
for i, k in enumerate(neighbors):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    # Compute training and test data accuracy
    train_accuracy[i] = knn.score(X_train, y_train)
    test_accuracy[i] = knn.score(X_test, y_test)

# Generate plot
plt.plot(neighbors, test_accuracy, label = 'Testing dataset
Accuracy')
plt.plot(neighbors, train_accuracy, label = 'Training datas
et Accuracy')

plt.legend()
plt.xlabel('n_neighbors')
plt.ylabel('Accuracy')
plt.show()
```



Here in the example shown above, we are creating a plot to see the k-value for which we have high accuracy.

Note: This is a technique which is not used industry-wide to choose the correct value of n_neighbors. Instead, we do hyperparameter tuning to choose the value that gives the best performance. We will be covering this in future posts.

Summary –

In this post, we have understood what supervised learning is and what are its categories. After having a basic understanding of Supervised learning we explored the k-nearest neighbor algorithm which is used to solve supervised machine learning problems. We also explored measuring the accuracy of the model.