SOC Mid-term Report

Introduction to Machine Learning

Jovita Bhasin (24B0939)
Mentor - Anuj Yadav

Pandas

Pandas is the most popular python library for data analysis.

Data Structures in Pandas

There are two core objects in pandas: the **DataFrame** and the **Series**.

DataFrame

A DataFrame is a table. It contains an array of individual entries, each of which has a certain value. Each entry corresponds to a row (or record) and a column
For example, consider the following DataFrame:

```
In [2]: pd.DataFrame({'Yes': [50, 21], 'No': [131, 2]})
```

Out[2]:

	Yes	No
0	50	131
1	21	2

In this example, the "0, No" entry has the value of 131. The "0, Yes" entry has a value of 50, and so on.

DataFrame entries are not limited to integers. For instance, here's a DataFrame whose entries are strings:

```
In [3]: pd.DataFrame({'Bob': ['I liked it.', 'It was awful.'], 'Sue': ['Pretty good.', 'Bland.']})
```

Out[3]:

	Bob	Sue
0	I liked it.	Pretty good.
1	It was awful.	Bland.

The list of row labels used in a DataFrame is known as an Index. We can assign values to it by using an index parameter in our constructor:

```
In [4]: pd.DataFrame({'Bob': ['I liked it.', 'It was awful.'],
                      'Sue': ['Pretty good.', 'Bland.']},
                     index=['Product A', 'Product B'])
```

Out[4]:

	Bob	Sue
Product A	I liked it.	Pretty good.
Product B	It was awful.	Bland.

Series

A Series, by contrast, is a sequence of data values. If a DataFrame is a table, a Series is a list. And in fact you can create one with nothing more than a list:

```
In [5]: pd.Series([1, 2, 3, 4, 5])

Out[5]: 0    1
        1    2
        2    3
        3    4
        4    5
        dtype: int64
```

A Series is, in essence, a single column of a DataFrame. So you can assign row labels to the Series the same way as before, using an index parameter. However, a Series does not have a column name, it only has one overall name.

The Series and the DataFrame are intimately related. It's helpful to think of a DataFrame as actually being just a bunch of Series "glued together".

Reading the data files

Being able to create a DataFrame or Series by hand is handy. But, most of the time, we won't actually be creating our own data by hand. Instead, we'll be working with data that already exists.
Data can be stored in any of a number of different forms and formats. By far the most basic of these is the CSV file.

We'll use the pd.read_csv() function to read the data into a DataFrame. This goes thusly:

```
In [7]: wine_reviews = pd.read_csv("../input/wine-reviews/winemag-data-130k-v2.csv")
```

We can use the shape attribute to check how large the resulting DataFrame is:

```
In [8]: wine_reviews.shape

Out[8]: (129971, 14)
```

We can examine the contents of the resultant DataFrame using the head() command, which grabs the first five rows.

Indexing, Selecting and Assigning

In Python, we can access the property of an object by accessing it as an attribute. A book object, for example, might have a title property, which we can access by calling book.title. Columns in a pandas DataFrame work in much the same way.

Hence to access the country property of reviews we can use:

```
In [3]: reviews.country

Out[3]: 0            Italy
        1         Portugal
                    ...
        129969      France
        129970      France
        Name: country, Length: 129971, dtype: object
```

If we have a Python dictionary, we can access its values using the indexing ([]) operator. We can do the same with columns in a DataFrame:

```
In [4]: reviews['country']

Out[4]: 0            Italy
        1         Portugal
                    ...
        129969      France
        129970      France
        Name: country, Length: 129971, dtype: object
```

These are the two ways of selecting a specific Series out of a DataFrame. Neither of them is more or less syntactically valid than the other, but the indexing operator [] does have the advantage that it can handle column names with reserved characters in them (e.g. if we had a country providence column, reviews.country providence wouldn't work).

To drill down to a single specific value, we need only use the indexing operator [] once more:

```
In [5]: reviews['country'][0]

Out[5]: 'Italy'
```

Pandas indexing works in one of two paradigms. The first is index-based selection: selecting data based on its numerical position in the data. iloc follows this paradigm.

To select the first row of data in a DataFrame, we may use the following:

```
In [6]: reviews.iloc[0]

Out[6]: country                                                 Italy
        description    Aromas include tropical fruit, broom, brimston...
                                                  ...
        variety                                          White Blend
        winery                                               Nicosia
        Name: 0, Length: 13, dtype: object
```

Both loc and iloc are row-first, column-second. This is the opposite of what we do in native Python, which is column-first, row-second.

The second paradigm for attribute selection is the one followed by the loc operator: label-based selection. In this paradigm, it's the data index value, not its position, which matters.

For example, to get the first entry in reviews, we would now do the following:

```
In [12]: reviews.loc[0, 'country']

Out[12]: 'Italy'
```

Choosing between loc and iloc

When choosing or transitioning between loc and iloc, there is one "gotcha" worth keeping in mind, which is that the two methods use slightly different indexing schemes.
iloc uses the Python stdlib indexing scheme, where the first element of the range is included and the last one excluded. So 0:10 will select entries 0,...,9. loc, meanwhile, indexes inclusively. So 0:10 will select entries 0,...,10.
Why the change? Remember that loc can index any stdlib type: strings, for example. If we have a DataFrame with index values Apples, ..., Potatoes, ..., and we want to select "all the alphabetical fruit choices between Apples and Potatoes", then it's a lot more convenient to index df.loc['Apples':'Potatoes'] than it is to index something like df.loc['Apples', 'Potatoet'] (t coming after s in the alphabet).
This is particularly confusing when the DataFrame index is a simple numerical list, e.g. 0,...,1000. In this case df.iloc[0:1000] will return 1000 entries, while df.loc[0:1000] return 1001 of them! To get 1000 elements using loc, you will need to go one lower and ask for df.loc[0:999].
Otherwise, the semantics of using loc are the same as those for iloc.

Conditional selection

So far we've been indexing various strides of data, using structural properties of the DataFrame itself. To do interesting things with the data, however, we often need to ask questions based on conditions.
For example, suppose that we're interested specifically in better-than-average wines produced in Italy.
We can start by checking if each wine is Italian or not:

```
In [15]: reviews.country == 'Italy'

Out[15]: 0          True
         1         False
                   ...
         129969    False
         129970    False
         Name: country, Length: 129971, dtype: bool
```

This operation produced a Series of True/False booleans based on the country of each record. This result can then be used inside of loc to select the relevant data.

Assigning data

Going the other way, assigning data to a DataFrame is easy. You can assign either a constant value:

```
In [21]:
reviews['critic'] = 'everyone'
reviews['critic']

Out[21]:
0         everyone
1         everyone
             ...
129969    everyone
129970    everyone
Name: critic, Length: 129971, dtype: object
```

Or with an iterable of values:

```
In [22]:
reviews['index_backwards'] = range(len(reviews), 0, -1)
reviews['index_backwards']

Out[22]:
0         129971
1         129970
             ...
129969         2
129970         1
Name: index_backwards, Length: 129971, dtype: int64
```

Summary Functions

Pandas provides many simple "summary functions" (not an official name) which restructure the data in some useful way. For example, consider the describe() method:

```
In [3]:
reviews.points.describe()

Out[3]:
count    129971.000000
mean         88.447138
                ...
75%          91.000000
max         100.000000
Name: points, Length: 8, dtype: float64
```

This method generates a high-level summary of the attributes of the given column. It is type-aware, meaning that its output changes based on the data type of the input. The output above only makes sense for numerical data; for string data here's what we get:

```
In [4]: reviews.taster_name.describe()

Out[4]: count         103727
        unique            19
        top       Roger Voss
        freq           25514
        Name: taster_name, dtype: object
```

If you want to get some particular simple summary statistic about a column in a DataFrame or a Series, there is usually a helpful pandas function that makes it happen.

For example, to see the mean of the points allotted (e.g. how well an averagely rated wine does), we can use the mean() function:

```
In [5]: reviews.points.mean()

Out[5]: 88.44713820775404
```

Groupwise Analysis

```
In [2]: reviews.groupby('points').points.count()

Out[2]: points
        80     397
        81     692
               ...
        99      33
        100     19
        Name: points, Length: 21, dtype: int64
```

groupby() created a group of reviews which allotted the same point values to the given wines. Then, for each of these groups, we grabbed the points() column and counted how many times it appeared.

Sorting

To get data in the order want it in we can sort it ourselves. The sort_values() method is handy for this.

```
In [10]: countries_reviewed = countries_reviewed.reset_index()
         countries_reviewed.sort_values(by='len')

Out[10]:
```

	country	province	len
179	Greece	Muscat of Kefallonian	1
192	Greece	Sterea Ellada	1
...	...	...	...
415	US	Washington	8639
392	US	California	36247

sort_values() defaults to an ascending sort, where the lowest values go first. However, most of the time we want a descending sort, where the higher numbers go first. That goes thusly:

```
In [11]: countries_reviewed.sort_values(by='len', ascending=False)
```

Out[11]:

	country	province	len
392	US	California	36247
415	US	Washington	8639
...	...	...	...
63	Chile	Coelemu	1
149	Greece	Beotia	1

425 rows × 3 columns

Finally, know that you can sort by more than one column at a time:

```
In [13]: countries_reviewed.sort_values(by=['country', 'len'])
```

Out[13]:

	country	province	len
1	Argentina	Other	536
0	Argentina	Mendoza Province	3264
...	...	...	...
424	Uruguay	Uruguay	24
419	Uruguay	Canelones	43

425 rows × 3 columns

Dtypes

The data type for a column in a DataFrame or a Series is known as the dtype.

You can use the dtype property to grab the type of a specific column. For instance, we can get the dtype of the price column in the reviews DataFrame:

```
In [2]: reviews.price.dtype
```

```
Out[2]: dtype('float64')
```

Alternatively, the dtypes property returns the dtype of every column in the DataFrame:

```
In [3]: reviews.dtypes
```

```
Out[3]: country        object
        description    object
                        ...
        variety        object
        winery         object
        Length: 13, dtype: object
```

Missing Data

Entries missing values are given the value NaN, short for "Not a Number". For technical reasons these NaN values are always of the float64 dtype.

Pandas provides some methods specific to missing data. To select NaN entries you can use pd.isnull() (or its companion pd.notnull()). This is meant to be used thusly:

```
In [6]: reviews[pd.isnull(reviews.country)]
```

Out[6]:

	country	description	designation	points	price	province	region_1	region_2	taster_name	taster_twitter_handle	title
913	NaN	Amber in color, this wine has aromas of peach ...	Asureti Valley	87	30.0	NaN	NaN	NaN	Mike DeSimone	@worldwineguys	Gotsa Family Wines 2014 Asureti Valley Chinuri
3131	NaN	Soft, fruity and juicy, this is a pleasant, si...	Partager	83	NaN	NaN	NaN	NaN	Roger Voss	@vossroger	Barton Guesti NV Partag Red
...	...	...	...	...	...	...	...	...	...	...	...
129590	NaN	A blend of 60% Syrah, 30% Cabernet Sauvignon a...	Shah	90	30.0	NaN	NaN	NaN	Mike DeSimone	@worldwineguys	Büyülü 2012 Shah F
129900	NaN	This wine offers a delightful bouquet of black...	NaN	91	32.0	NaN	NaN	NaN	Mike DeSimone	@worldwineguys	Psagot 2014 Merlot

Replacing missing values is a common operation. Pandas provides a really handy method for this problem: fillna(). fillna() provides a few different strategies for mitigating such data. For example, we can simply replace each NaN with an "Unknown":

```
In [7]: reviews.region_2.fillna("Unknown")
```

Out[7]:

```
0          Unknown
1          Unknown
              ...
129969     Unknown
129970     Unknown
Name: region_2, Length: 129971, dtype: object
```

Or we could fill each missing value with the first non-null value that appears sometime after the given record in the database. This is known as the backfill strategy.

Alternatively, we may have a non-null value that we would like to replace. For example, suppose that since this dataset was published, reviewer Kerin O'Keefe has changed her Twitter handle from @kerinokeefe to @kerino. One way to reflect this in the dataset is using the replace() method:

```
In [8]: reviews.taster_twitter_handle.replace("@kerinokeefe", "@kerino")

Out[8]: 0            @kerino
        1          @vossroger
                      ...
        129969     @vossroger
        129970     @vossroger
        Name: taster_twitter_handle, Length: 129971, dtype: object
```

MACHINE LEARNING

How models work?

Machine learning models work by identifying patterns in data and using those patterns to make predictions or decisions without being explicitly programmed for every task. At their core, these models learn from examples — they are trained on datasets where the correct outputs are known, allowing them to adjust their internal parameters to minimize errors. Once trained, the model can generalize this learned behavior to new, unseen data. This approach allows machines to adapt and improve over time, making machine learning a powerful tool for applications ranging from image recognition and natural language processing to medical diagnosis and recommendation systems.

Supervised Machine Learning

Supervised machine learning is a type of machine learning where the model is trained on a labeled dataset — meaning that each training example is paired with the correct output (also called the "label"). The goal of the model is to learn a mapping from inputs to outputs so that it can predict the label for new, unseen data. For example, in a spam detection system, the input might be an email, and the label could be "spam" or "not spam."

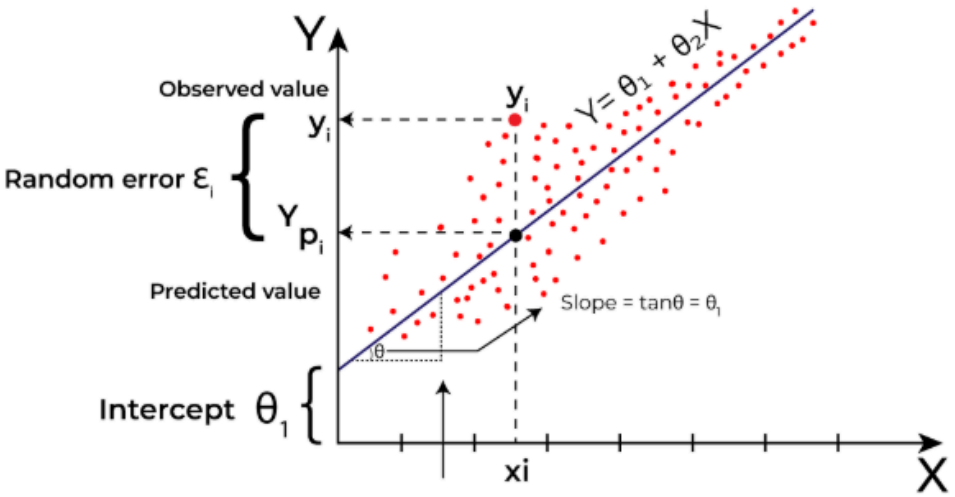
Linear Regression

Linear regression is a type of supervised machine learning algorithm that learns from the labelled datasets and maps the data points with most optimized linear functions which can be used for prediction on new datasets. It assumes that there is a linear relationship between the input and output, meaning the output changes at a constant rate as the input changes. This relationship is represented by a straight line.

In linear regression, the best-fit line is the straight line that most accurately represents the relationship between the independent variable (input) and the dependent variable (output). It is the line that minimizes the difference between the actual data points and the predicted values from the model.

1. Goal of the Best-Fit Line

The goal of linear regression is to find a straight line that minimizes the error (the difference) between the observed data points and the predicted values. This line helps us predict the dependent variable for new, unseen data.



Linear Regression

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.

2. Equation of the Best-Fit Line

For simple linear regression (with one independent variable), the best-fit line is represented by the equation:

$$y = mx + b$$

Where:

- y is the predicted value (dependent variable)
- x is the input (independent variable)
- m is the slope of the line (how much y changes when x changes)
- b is the intercept (the value of y when x = 0)

The best-fit line will be the one that optimizes the values of m (slope) and b (intercept) so that the predicted y values are as close as possible to the actual data points.

3. Minimizing the Error: The Least Squares Method

To find the best-fit line, we use a method called Least Squares. The idea behind this method is to minimize the sum of squared differences between the actual values (data points) and the predicted values from the line. These differences are called residuals.

$$Residual = y_i - \hat{y}_i$$

Where:

- y_i is the actual observed value
- $\hat{y}_i$ is the predicted value from the line for that x_i

The least squares method minimizes the sum of the squared residuals:

$$Sum of squared errors (SSE) = \Sigma (y_i - \hat{y}_i)^2$$

This method ensures that the line best represents the data where the sum of the squared differences between the predicted values and actual values is as small as possible.

4. Interpretation of the Best-Fit Line

- Slope (m): The slope of the best-fit line indicates how much the dependent variable (y) changes with each unit change in the independent variable (x). For example if the slope is 5, it means that for every 1-unit increase in x, the value of y increases by 5 units.
- Intercept (b): The intercept represents the predicted value of y when x = 0. It's the point where the line crosses the y-axis.

5. Cost function for Linear Regression

MSE function can be calculated as:

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

Our goal is to minimize the cost function.

6. Gradient Descent for Linear Regression

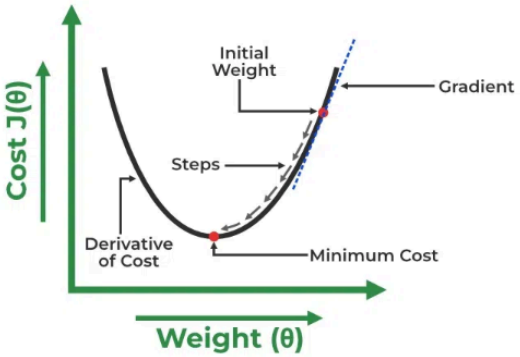
A linear regression model can be trained using the optimization algorithm gradient descent by iteratively modifying the model's parameters to reduce the mean squared error (MSE) of the model on a training dataset. To update θ1 and θ2 values in order to reduce the Cost function (minimizing RMSE value)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

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

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

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 α is the learning rate.



Gradient Descent

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

Regularization Techniques for Linear Models

1. Lasso Regression (L1 Regularization)

Lasso Regression is a technique used for regularizing a linear regression model, it adds a penalty term to the linear regression objective function to prevent overfitting.
The objective function after applying lasso regression is:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(\widehat{y_i} - y_i)^2 + \lambda\sum_{j=1}^{n}|\theta_j|$$

- the first term is the least squares loss, representing the squared difference between predicted and actual values.
- the second term is the L1 regularization term, it penalizes the sum of absolute values of the regression coefficient θj.

2. Ridge Regression (L2 Regularization)

Ridge Regression is a linear regression technique that adds a regularization term to the standard linear objective. Again, the goal is to prevent overfitting by penalizing large coefficient in linear regression equation. It useful when the dataset has multicollinearity where predictor variables are highly correlated.
The objective function after applying ridge regression is:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(\widehat{y_i} - y_i)^2 + \lambda\sum_{j=1}^{n}\theta_j^2$$

- the first term is the least squares loss, representing the squared difference between predicted and actual values.
- the second term is the L1 regularization term, it penalizes the sum of square of values of the regression coefficient θj.

One Hot Encoding

One Hot Encoding is a method for converting categorical variables into a binary format. It creates new columns for each category where 1 means the category is present and 0 means it is not. The primary purpose of One Hot Encoding is to ensure that categorical data can be effectively used in machine learning models.

How One-Hot Encoding Works: An Example

To grasp the concept better let's explore a simple example. Imagine we have a dataset with fruits, their categorical values and corresponding prices. Using one-hot encoding we can transform these categorical values into numerical form. For example:

- Wherever the fruit is "Apple," the Apple column will have a value of 1 while the other fruit columns (like Mango or Orange) will contain 0.
- This pattern ensures that each categorical value gets its own column represented with binary values (1 or 0) making it usable for machine learning models.

Fruit	Categorical value of fruit	Price
apple	1	5
mango	2	10
apple	1	15
orange	3	20

The output after applying one-hot encoding on the data is given as follows,

Fruit_apple	Fruit_mango	Fruit_orange	price
1	0	0	5
0	1	0	10
1	0	0	15
0	0	1	20

sklearn.model_selection.train_test_split() function:

The train_test_split() method is used to split our data into train and test sets. First, we need to divide our data into features (X) and labels (y). The dataframe gets divided into X_train, X_test, y_train, and y_test. X_train and y_train sets are used for training and fitting the model. The X_test and y_test sets are used for testing the model if it's predicting the right outputs/labels. we can explicitly test the size of the train and test sets. It is suggested to keep our train sets larger than the test sets.

- Train set: The training dataset is a set of data that was utilized to fit the model. The dataset on which the model is trained. This data is seen and learned by the model.
- Test set: The test dataset is a subset of the training dataset that is utilized to give an accurate evaluation of a final model fit.
- validation set: A validation dataset is a sample of data from your model's training set that is used to estimate model performance while tuning the model's hyperparameters.
- underfitting: A data model that is under-fitted has a high error rate on both the training set and unobserved data because it is unable to effectively represent the relationship between the input and output variables.
- overfitting: when a statistical model matches its training data exactly but the algorithm's goal is lost because it is unable to accurately execute against unseen data is called overfitting

Logistic Regression

Logistic Regression is a supervised machine learning algorithm used for classification problems. Unlike linear regression which predicts continuous values it predicts the probability that an input belongs to a specific class. It is used for binary classification where the output can be one of two possible categories such as Yes/No, True/False or 0/1. It uses sigmoid function to convert inputs into a probability value between 0 and 1.

Types of Logistic Regression

Logistic regression can be classified into three main types based on the nature of the dependent variable:

1. Binomial Logistic Regression: This type is used when the dependent variable has only two possible categories. Examples include Yes/No, Pass/Fail or 0/1. It is the most common form of logistic regression and is used for binary classification problems.
2. Multinomial Logistic Regression: This is used when the dependent variable has three or more possible categories that are not ordered. For example, classifying animals into categories like "cat," "dog" or "sheep." It extends the binary logistic regression to handle multiple classes.
3. Ordinal Logistic Regression: This type applies when the dependent variable has three or more categories with a natural order or ranking. Examples include ratings like "low," "medium" and "high." It takes the order of the categories into account when modeling.

Understanding Sigmoid Function

1. The sigmoid function is a important part of logistic regression which is used to convert the raw output of the model into a probability value between 0 and 1.
2. This function takes any real number and maps it into the range 0 to 1 forming an "S" shaped curve called the sigmoid curve or logistic curve. Because probabilities must lie between 0 and 1, the sigmoid function is perfect for this purpose.
3. In logistic regression, we use a threshold value usually 0.5 to decide the class label.
   - If the sigmoid output is same or above the threshold, the input is classified as Class 1.
   - If it is below the threshold, the input is classified as Class 0.

This approach helps to transform continuous input values into meaningful class predictions.

How does Logistic Regression work?

Logistic regression model transforms the linear regression function continuous value output into categorical value output using a sigmoid function which maps any real-valued set of independent variables input into a value between 0 and 1. This function is known as the logistic function.
Suppose we have input features represented as a matrix:

$$X = \begin{bmatrix} x_{11} & \cdots & x_{1m} \\ x_{21} & \cdots & x_{2m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nm} \end{bmatrix}$$

and the dependent variable is Y having only binary value i.e 0 or 1.

$$Y = \begin{cases} 0 & \text{if } Class\ 1 \\ 1 & \text{if } Class\ 2 \end{cases}$$

then, apply the multi-linear function to the input variables X.

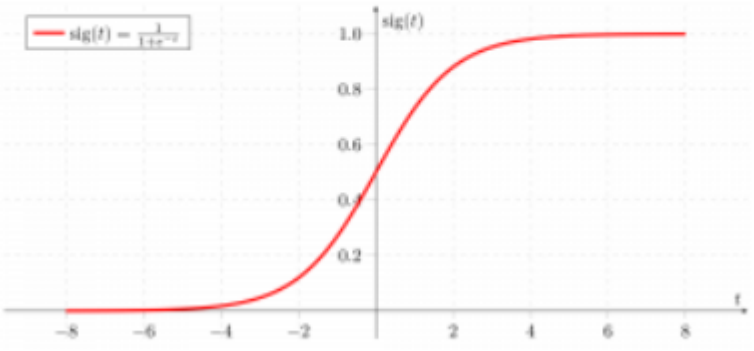
$$z = (\sum_{i=1}^{n} w_i x_i) + b$$

Here x_i is the ith observation of X, $w_i = [w_1, w_2, w_3, \cdots, w_m]$ is the weights or Coefficient and $b$is the bias term also known as intercept. Simply this can be represented as the dot product of weight and bias.

$$z = w \cdot X + b$$

At this stage, z is a continuous value from the linear regression. Logistic regression then applies the sigmoid function to z to convert it into a probability between 0 and 1 which can be used to predict the class.
Now we use the sigmoid function where the input will be z and we find the probability between 0 and 1. i.e. predicted y.

$$\sigma(z) = \frac{1}{1+e^{-z}}$$



Sigmoid function

As shown above the sigmoid function converts the continuous variable data into the probability i.e between 0 and 1.

where the probability of being a class can be measured as:

$$P(y = 1) = \sigma(z)$$
$$P(y = 0) = 1 - \sigma(z)$$

Decision Tree

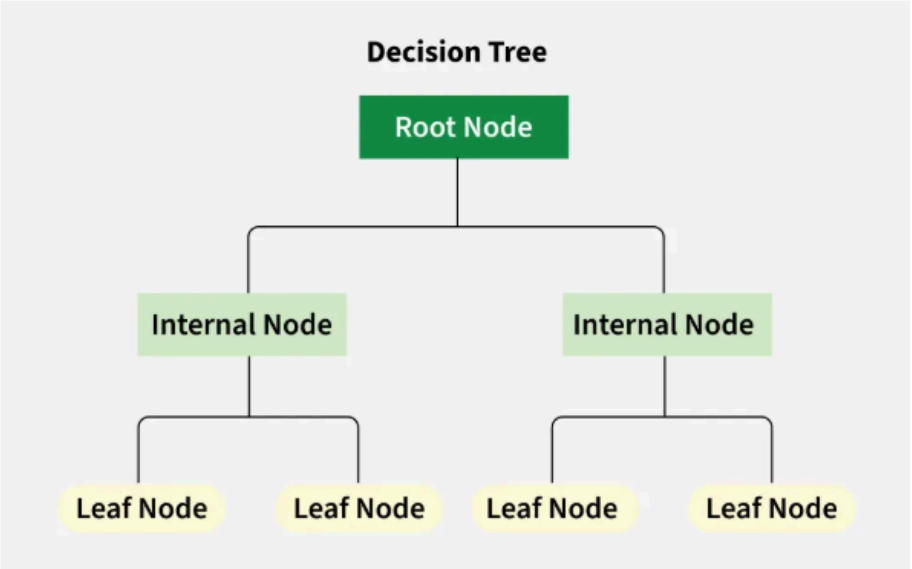
Decision tree is a simple diagram that shows different choices and their possible results helping you make decisions easily.

Understanding Decision Tree

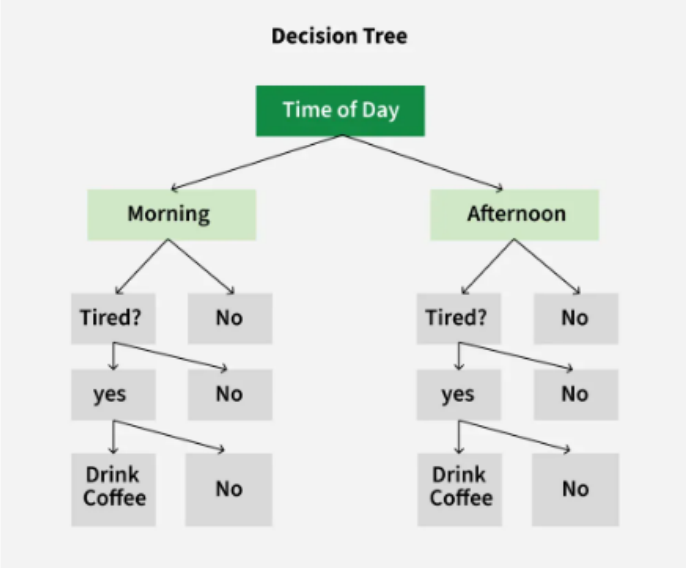
A decision tree is a graphical representation of different options for solving a problem and show how different factors are related. It has a hierarchical tree structure starts with one

main question at the top called a node which further branches out into different possible outcomes where:

- Root Node is the starting point that represents the entire dataset.
- Branches: These are the lines that connect nodes. It shows the flow from one decision to another.
- Internal Nodes are Points where decisions are made based on the input features.
- Leaf Nodes: These are the terminal nodes at the end of branches that represent final outcomes or predictions



Now, let's take an example to understand the decision tree. Imagine you want to decide whether to drink coffee based on the time of day and how tired you feel. First the tree checks the time of day—if it's morning it asks whether you are tired. If you're tired the tree suggests drinking coffee if not it says there's no need. Similarly in the afternoon the tree again asks if you are tired. If you recommend drinking coffee if not it concludes no coffee is needed.



How Decision Trees Work?

A decision tree working starts with a main question known as the root node. This question is derived from the features of the dataset and serves as the starting point for decision-making. From the root node, the tree asks a series of yes/no questions. Each question is designed to split the data into subsets based on specific attributes. For example if the first question is "Is it raining?", the answer will determine which branch of the tree to follow. Depending on the response to each question you follow different branches. If your answer is "Yes," you might proceed down one path if "No," you will take another path.
This branching continues through a sequence of decisions. As you follow each branch, you get more questions that break the data into smaller groups. This step-by-step process continues until you have no more helpful questions .
You reach at the end of a branch where you find the final outcome or decision. It could be a classification (like "spam" or "not spam") or a prediction (such as estimated price).

Advantages of Decision Trees

- Simplicity and Interpretability: Decision trees are straightforward and easy to understand. You can visualize them like a flowchart which makes it simple to see how decisions are made.
- Versatility: It means they can be used for different types of tasks can work well for both classification and regression
- No Need for Feature Scaling: They don’t require you to normalize or scale your data.
- Handles Non-linear Relationships: It is capable of capturing non-linear relationships between features and target variables.

Disadvantages of Decision Trees

- Overfitting: Overfitting occurs when a decision tree captures noise and details in the training data and it perform poorly on new data.
- Instability: instability means that the model can be unreliable slight variations in input can lead to significant differences in predictions.
- Bias towards Features with More Levels: Decision trees can become biased towards features with many categories focusing too much on them during decision-making. This can cause the model to miss out other important features led to less accurate predictions .

Support Vector Machine (svm)

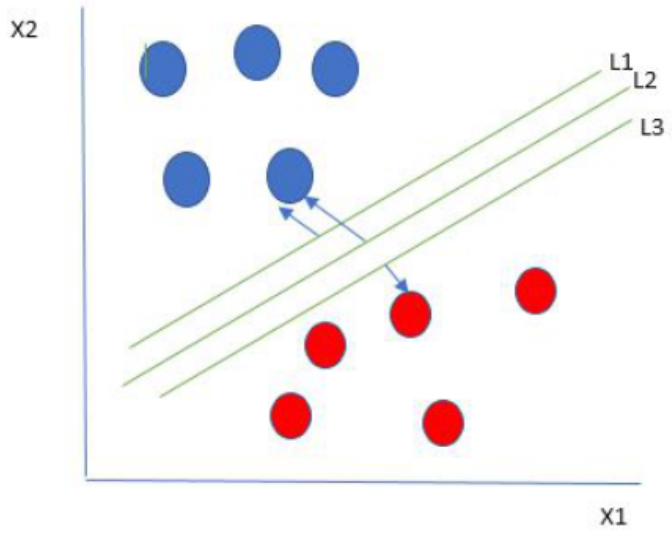
Support Vector Machine (SVM) is a supervised machine learning algorithm used for classification and regression tasks. It tries to find the best boundary known as hyperplane that separates different classes in the data. It is useful when you want to do binary classification like spam vs. not spam or cat vs. dog.
The main goal of SVM is to maximize the margin between the two classes. The larger the margin the better the model performs on new and unseen data.

Key Concepts of Support Vector Machine

- Hyperplane: A decision boundary separating different classes in feature space and is represented by the equation wx + b = 0 in linear classification.
- Support Vectors: The closest data points to the hyperplane, crucial for determining the hyperplane and margin in SVM.
- Margin: The distance between the hyperplane and the support vectors. SVM aims to maximize this margin for better classification performance.
- Kernel: A function that maps data to a higher-dimensional space enabling SVM to handle non-linearly separable data.
- Hard Margin: A maximum-margin hyperplane that perfectly separates the data without misclassifications.
- Soft Margin: Allows some misclassifications by introducing slack variables, balancing margin maximization and misclassification penalties when data is not perfectly separable.
- C: A regularization term balancing margin maximization and misclassification penalties. A higher C value forces stricter penalty for misclassifications.
- Hinge Loss: A loss function penalizing misclassified points or margin violations and is combined with regularization in SVM.
- Dual Problem: Involves solving for Lagrange multipliers associated with support vectors, facilitating the kernel trick and efficient computation.

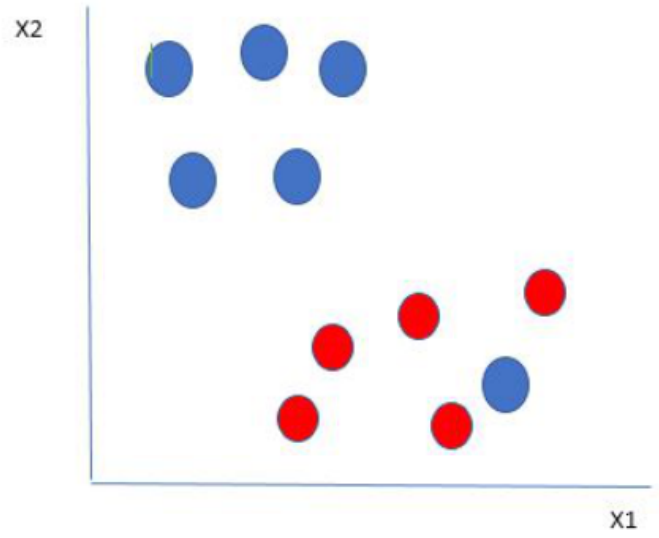
How does Support Vector Machine Algorithm Work?

The key idea behind the SVM algorithm is to find the hyperplane that best separates two classes by maximizing the margin between them. This margin is the distance from the hyperplane to the nearest data points (support vectors) on each side.



Multiple hyperplanes separate the data from two classes

The best hyperplane also known as the "hard margin" is the one that maximizes the distance between the hyperplane and the nearest data points from both classes. This ensures a clear separation between the classes. So from the above figure, we choose L2 as hard margin. Let's consider a scenario like shown below:

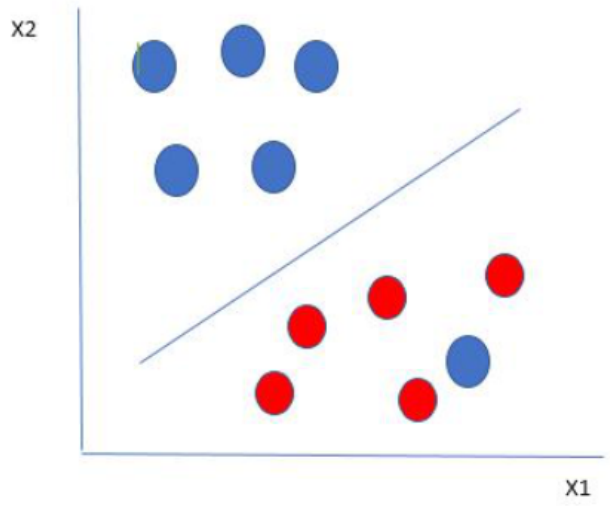


Selecting hyperplane for data with outlier

Here, we have one blue ball in the boundary of the red ball.

How does SVM classify the data?

The blue ball in the boundary of red ones is an outlier of blue balls. The SVM algorithm has the characteristics to ignore the outlier and finds the best hyperplane that maximizes the margin. SVM is robust to outliers.



Hyperplane which is the most optimized one

A soft margin allows for some misclassifications or violations of the margin to improve generalization. The SVM optimizes the following equation to balance margin maximization and penalty minimization:

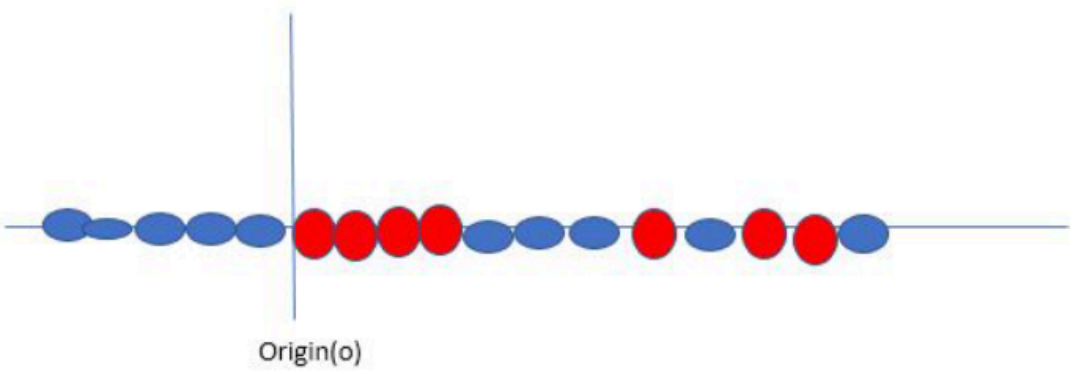
$$\text{Objective Function} = \left(\frac{1}{\text{margin}}\right) + \lambda \sum \text{penalty}$$

The penalty used for violations is often hinge loss which has the following behavior:

- If a data point is correctly classified and within the margin there is no penalty (loss = 0).
- If a point is incorrectly classified or violates the margin the hinge loss increases proportionally to the distance of the violation.

What to do if data are not linearly separable?

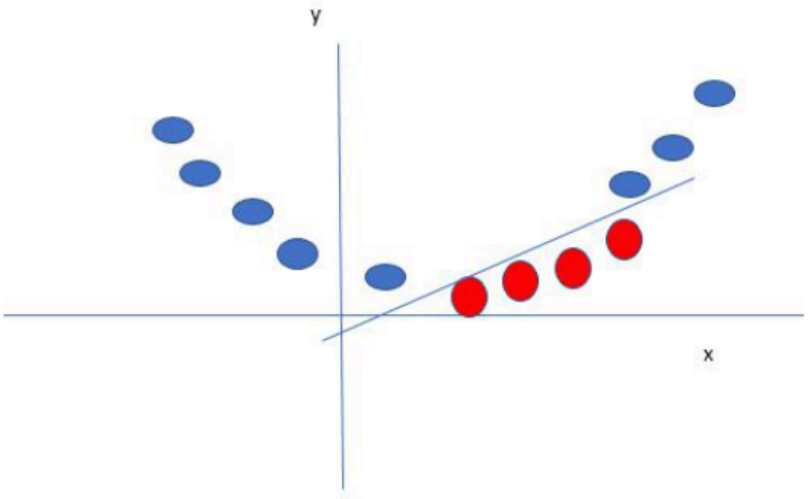
When data is not linearly separable i.e it can't be divided by a straight line, SVM uses a technique called kernels to map the data into a higher-dimensional space where it becomes separable. This transformation helps SVM find a decision boundary even for non-linear data.



Original 1D dataset for classification

A kernel is a function that maps data points into a higher-dimensional space without explicitly computing the coordinates in that space. This allows SVM to work efficiently with non-linear data by implicitly performing the mapping. For example consider data points that are not linearly separable. By applying a kernel function SVM transforms the data points into a higher-dimensional space where they become linearly separable.

- Linear Kernel: For linear separability.
- Polynomial Kernel: Maps data into a polynomial space.
- Radial Basis Function (RBF) Kernel: Transforms data into a space based on distances between data points.



Mapping 1D data to 2D to become able to separate the two classes

In this case the new variable y is created as a function of distance from the origin.

Mathematical Computation of SVM

Consider a binary classification problem with two classes, labeled as +1 and -1. We have a training dataset consisting of input feature vectors X and their corresponding class labels Y. The equation for the linear hyperplane can be written as:

$$w^T x + b = 0$$

Where:

- w is the normal vector to the hyperplane (the direction perpendicular to it).
- b is the offset or bias term representing the distance of the hyperplane from the origin along the normal vector

Distance from a Data Point to the Hyperplane

The distance between a data point x_i and the decision boundary can be calculated as:

$$d_i = \frac{w^T x_i + b}{||w||}$$

where ||w|| represents the Euclidean norm of the weight vector w. Euclidean norm of the normal vector W

Linear SVM Classifier

Distance from a Data Point to the Hyperplane:

$$\hat{y} = \begin{cases} 1 & : \ w^T x + b \geq 0 \\ 0 & : \ w^T x + b < 0 \end{cases}$$

Where y^ is the predicted label of a data point.

Optimization Problem for SVM

For a linearly separable dataset the goal is to find the hyperplane that maximizes the margin between the two classes while ensuring that all data points are correctly classified. This leads to the following optimization problem:

$$\underset{w,b}{\text{minimize}} \frac{1}{2} \|w\|^2$$

Subject to the constraint:

$$y_i(w^T x_i + b) \geq 1 \ for \ i = 1, 2, 3, \cdots, m$$

Where:

- yi is the class label (+1 or -1) for each training instance.

- xi is the feature vector for the i-th training instance.
- m is the total number of training instances.

Types of Support Vector Machine

Based on the nature of the decision boundary, Support Vector Machines (SVM) can be divided into two main parts:

- Linear SVM: Linear SVMs use a linear decision boundary to separate the data points of different classes. When the data can be precisely linearly separated, linear SVMs are very suitable. This means that a single straight line (in 2D) or a hyperplane (in higher dimensions) can entirely divide the data points into their respective classes. A hyperplane that maximizes the margin between the classes is the decision boundary.
- Non-Linear SVM: can be used to classify data when it cannot be separated into two classes by a straight line (in the case of 2D). By using kernel functions, nonlinear SVMs can handle nonlinearly separable data. The original input data is transformed by these kernel functions into a higher-dimensional feature space where the data points can be linearly separated. A linear SVM is used to locate a nonlinear decision boundary in this modified space.

Random Forest Algorithm

Random Forest is a machine learning algorithm that uses many decision trees to make better predictions. Each tree looks at different random parts of the data and their results are combined by voting for classification or averaging for regression. This helps in improving accuracy and reducing errors.

Working of Random Forest Algorithm

- Create Many Decision Trees: The algorithm makes many decision trees each using a random part of the data. So every tree is a bit different.
- Pick Random Features: When building each tree it doesn't look at all the features (columns) at once. It picks a few at random to decide how to split the data. This helps the trees stay different from each other.
- Each Tree Makes a Prediction: Every tree gives its own answer or prediction based on what it learned from its part of the data.
- Combine the Predictions:
    - For classification we choose a category as the final answer is the one that most trees agree on i.e majority voting.
    - For regression we predict a number as the final answer is the average of all the trees predictions.
- Why It Works Well: Using random data and features for each tree helps avoid overfitting and makes the overall prediction more accurate and trustworthy.

Key Features of Random Forest

- Handles Missing Data: It can work even if some data is missing so you don't always need to fill in the gaps yourself.

- Shows Feature Importance: It tells you which features (columns) are most useful for making predictions which helps you understand your data better.
- Works Well with Big and Complex Data: It can handle large datasets with many features without slowing down or losing accuracy.
- Used for Different Tasks: You can use it for both classification like predicting types or labels and regression like predicting numbers or amounts.

Assumptions of Random Forest

- Each tree makes its own decisions: Every tree in the forest makes its own predictions without relying on others.
- Random parts of the data are used: Each tree is built using random samples and features to reduce mistakes.
- Enough data is needed: Sufficient data ensures the trees are different and learn unique patterns and variety.
- Different predictions improve accuracy: Combining the predictions from different trees leads to a more accurate final result.

K-Fold Cross Validation

In K-Fold Cross Validation we split the dataset into k number of subsets known as folds then we perform training on the all the subsets but leave one (k-1) subset for the evaluation of the trained model. In this method, we iterate k times with a different subset reserved for testing purpose each time.

Naive Bayes Classifier

Naive Bayes is a probabilistic classification algorithm based on Bayes' Theorem. It assumes that the features used to predict the output are independent of each other — an assumption that is rarely true in practice, but often works surprisingly well in real-world applications.
The algorithm calculates the posterior probability for each class, given the input features, and predicts the class with the highest probability. Despite its simplicity, Naive Bayes is fast, efficient, and performs well on high-dimensional data.
It is commonly used in text classification problems, such as spam detection, sentiment analysis, and document categorization, where feature independence is approximately valid.

Types of Naive Bayes Model

There are three types of Naive Bayes Model :

1. Gaussian Naive Bayes

In Gaussian Naive Bayes, continuous values associated with each feature are assumed to be distributed according to a Gaussian distribution. A Gaussian distribution is also called Normal distribution. When plotted, it gives a bell shaped curve which is symmetric about the mean of the feature values as shown below:

2. Multinomial Naive Bayes

It is used when features represent the frequency of terms (such as word counts) in a document. It is commonly applied in text classification, where term frequencies are important.

3. Bernoulli Naive Bayes

It deals with binary features, where each feature indicates whether a word appears or not in a document. It is suited for scenarios where the presence or absence of terms is more relevant than their frequency. Both models are widely used in document classification tasks

Unsupervised Machine Learning

Unsupervised machine learning is a type of learning where the model is trained on data without any labeled outputs. Instead of learning from known input-output pairs, the model tries to identify patterns, structures, or groupings within the data on its own. It is commonly used for tasks such as clustering, where similar data points are grouped together, and dimensionality reduction, where complex data is simplified while preserving essential information.

K means clustering

K-Means Clustering is an Unsupervised Machine Learning algorithm which groups unlabeled dataset into different clusters. It is used to organize data into groups based on their similarity.

Steps of K-Means Clustering

1. Choose the number of clusters (k):
   Decide how many clusters you want the algorithm to form.

2. Initialize centroids:
   Randomly select k data points as the initial cluster centroids (center points).

3. Assign points to nearest centroid:
   For each data point, calculate its distance to all centroids and assign it to the nearest one. This forms k clusters.

4. Update centroids:
   For each cluster, recalculate the centroid by taking the mean of all points in that cluster.

5. Repeat steps 3–4:
   Reassign points based on the new centroids and update the centroids again. Repeat this process until the cluster assignments no longer change or until a maximum number of iterations is reached.

6. Final clusters:
   Once convergence is reached, the final centroids and their corresponding data points represent the final clusters.

Hierarchical Clustering

Hierarchical clustering is used to group similar data points together based on their similarity creating a hierarchy or tree-like structure. The key idea is to begin with each data point as its own separate cluster and then progressively merge or split them based on their similarity.

Steps of Hierarchical Clustering

Hierarchical clustering builds a tree-like structure of clusters called a dendrogram. There are two main types: Agglomerative (bottom-up) and Divisive (top-down). The most commonly used is Agglomerative Hierarchical Clustering, and its steps are:

1. Start with each data point as its own cluster:
   If there are n data points, start with n individual clusters.

2. Compute distance (similarity) between all clusters:
   Use a distance metric (like Euclidean distance) to calculate how close each pair of clusters is.

3. Merge the two closest clusters:
   Combine the pair of clusters that are closest together to form a new cluster.

4. Recompute distances between the new cluster and all others:
   Update the distance matrix based on the chosen linkage method (e.g., single, complete, average, or Ward's method).

5. Repeat steps 3–4:
   Continue merging the closest clusters and updating distances until all data points are in one single cluster (the root of the dendrogram).

6. Decide the number of clusters (optional):
   Cut the dendrogram at the desired level (height) to form a specific number of clusters.

DBSCAN Clustering

DBSCAN is a density-based clustering algorithm that groups data points that are closely packed together and marks outliers as noise based on their density in the feature space. It identifies clusters as dense regions in the data space separated by areas of lower density. Unlike K-Means or hierarchical clustering which assumes clusters are compact and spherical, DBSCAN perform well in handling real-world data irregularities such as:

- Arbitrary-Shaped Clusters: Clusters can take any shape not just circular or convex.
- Noise and Outliers: It effectively identifies and handles noise points without assigning them to any cluster.

Key Parameters in DBSCAN

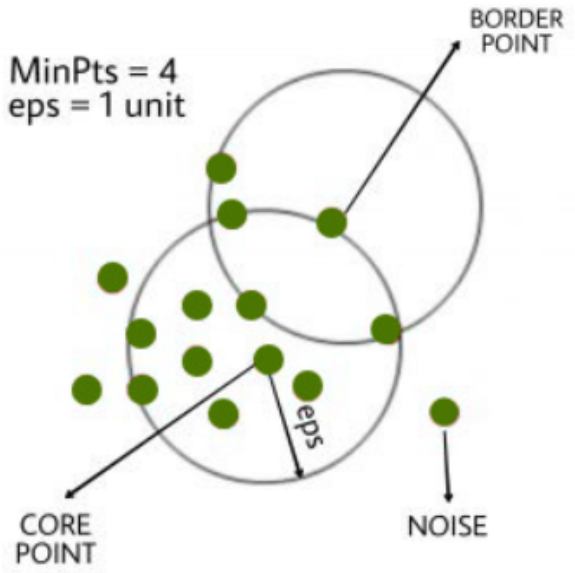
1. eps: This defines the radius of the neighborhood around a data point. If the distance between two points is less than or equal to eps they are considered neighbors. A common method to determine eps is by analyzing the k-distance graph. Choosing the right eps is important:
   - If eps is too small most points will be classified as noise.
   - If eps is too large clusters may merge and the algorithm may fail to distinguish between them.
2. MinPts: This is the minimum number of points required within the eps radius to form a dense region. A general rule of thumb is to set MinPts >= D+1 where D is the number of dimensions in the dataset.

How Does DBSCAN Work?

DBSCAN works by categorizing data points into three types:

1. Core points which have a sufficient number of neighbors within a specified radius (eplison)
2. Border points which are near core points but lack enough neighbors to be core points themselves
3. Noise points which do not belong to any cluster.

By iteratively expanding clusters from core points and connecting density-reachable points, DBSCAN forms clusters without relying on rigid assumptions about their shape or size.



Steps in the DBSCAN Algorithm

1. Identify Core Points: For each point in the dataset count the number of points within its eps neighborhood. If the count meets or exceeds MinPts mark the point as a core point.

2. Form Clusters: For each core point that is not already assigned to a cluster create a new cluster. Recursively find all density-connected points i.e points within the eps radius of the core point and add them to the cluster.
3. Density Connectivity: Two points a and b are density-connected if there exists a chain of points where each point is within the eps radius of the next and at least one point in the chain is a core point. This chaining process ensures that all points in a cluster are connected through a series of dense regions.
4. Label Noise Points: After processing all points any point that does not belong to a cluster is labeled as noise.