Train Test Split:

The train-test split is a technique for evaluating the performance of a machine learning algorithm.

It can be used for classification or regression problems and can be used for any supervised learning algorithm.

The procedure involves taking the dataset and splitting it into 2 subsets.

- **Train Dataset**: Used to fit the machine learning model.
- **Test Dataset**: Used to evaluate the fit machine learning model.

The objective is to estimate the performance of the machine learning model on new data: data not used to train the model.

When to Use the Train-Test Split:

When the dataset is sufficiently large and each of the train and test datasets are suitable representations of the problem domain.

The train-test split evaluation procedure is computational efficiency

the train-test split procedure is used to verify the performance of the model quickly across vast data already present.

How to Configure the Train-Test Split

one main configuration parameter, which is the **size of the train and test sets**

You can split based on your project objectives:

- Computational cost in training the model.
- Computational cost in evaluating the model.
- Training set representativeness.
- Test set representativeness.

Commonly used splits:

- Train: 80%, Test: 20%
- Train: 67%, Test: 33%
- Train: 50%, Test: 50%

Stratified Train-Test Splits:

Some classification problems do not have a balanced number of examples for each class label.

As such, it is desirable to split the dataset into train and test sets in a way that preserves the same proportions of examples in each class as observed in the original dataset.

This is called a **stratified train-test split.**

We can achieve this by setting the "***stratify***" argument to the y component of the original dataset. This will be used by the *train_test_split()* function to ensure that both the train and test sets have the proportion of examples in each class that is present in the provided "*y*" array.

Classification:

Logistic Regression

Logistic Regression is a classification technique used in machine learning. It uses a logistic function to model the dependent variable.

Types of Logistic Regression:

Though generally used for predicting binary target variables, logistic regression can be extended and further classified into three different types that are as mentioned below:

1. **Binomial**: Where the target variable can have only two possible types. **eg.**: Predicting a mail as spam or not.
2. **Multinomial**: Where the target variable has three or more possible types, which may not have any quantitative significance. **eg.**: Predicting disease.
3. **Ordinal**: Where the target variables have ordered categories. **eg.**: Web Series ratings from 1 to 5.

In logistic regression in order to map the predicted values to probabilities, the sigmoid function is used. This function maps any real value into another value between **0 to 1**

SIGMOID:

Formula

$$S(x) = \frac{1}{1 + e^{-x}}$$

$S(x)$ = sigmoid function

e = Euler's number

Cost Function

Logistic regression uses Log loss or cross-entropy.

$$\text{Cost}(h_\Theta(x), Y(\text{actual})) = -\log(h_\Theta(x)) \text{ if } y=1$$
$$-\log(1- h_\Theta(x)) \text{ if } y=0$$

#sample Logistic regression code:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.20, random_state = 99)

from sklearn.linear_model import LogisticRegression

model = LogisticRegression()

model.fit(X_train, y_train)

model.score(X_test,y_test)
```

Naive Bayes:

t is a classification technique based on Bayes' Theorem with an assumption of independence among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature.

Bayes Theorem:

Formula

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

A, B = events

$P(A|B)$ = probability of A given B is true

$P(B|A)$ = probability of B given A is true

$P(A), P(B)$ = the independent probabilities of A and B

How Naive Bayes algorithm works?

Let's consider a training data set of weather and corresponding target variable 'Play' Now, we need to classify whether players will play or not based on weather conditions.

Step 1: Convert the data set into a frequency table

Step 2: Create a Likelihood table by finding the probabilities like Overcast probability = 0.29 and probability of playing is 0.64.

Weather	Play
Sunny	No
Overcast	Yes
Rainy	Yes
Sunny	Yes
Sunny	Yes
Overcast	Yes
Rainy	No
Rainy	No
Sunny	Yes
Rainy	Yes
Sunny	No
Overcast	Yes
Overcast	Yes
Rainy	No

Frequency Table		
Weather	No	Yes
Overcast		4
Rainy	3	2
Sunny	2	3
Grand Total	5	9

Likelihood table				
Weather	No	Yes		
Overcast		4	=4/14	0.29
Rainy	3	2	=5/14	0.36
Sunny	2	3	=5/14	0.36
All	5	9		
	=5/14	=9/14		
	0.36	0.64		

Step 3: Now, use a Naive Bayesian equation to calculate the posterior probability for each class.

Problem: Players will play if the weather is sunny. Is this statement is correct?

We can solve it using above discussed method of the posterior probability.

P(Yes | Sunny) = P(Sunny | Yes) * P(Yes) / P (Sunny)

Here we have P (Sunny |Yes) = 3/9 = 0.33, P(Sunny) = 5/14 = 0.36, P(Yes)= 9/14 = 0.64

Now, P (Yes | Sunny) = 0.33 * 0.64 / 0.36 = 0.60, which has higher probability.

CODE OF DECISION TREE:

```
from sklearn.tree import DecisionTreeClassifier

clf_model = DecisionTreeClassifier(criterion="gini",
random_state=42,max_depth=3, min_samples_leaf=5)

#train

clf_model.fit(X_train,y_train)

#predict

y_predict = clf_model.predict(X_test)

#metrics

from sklearn.metrics import
accuracy_score,classification_report,confusion_matrix

accuracy_score(y_test,y_predict)

#plot decision tree

from sklearn import tree

import graphviz

dot_data = tree.export_graphviz(clf_model,

                                out_file=None,

                      feature_names=feature_names,
```

```
                        class_names=target,

                        filled=True, rounded=True,

                        special_characters=True)

graph = graphviz.Source(dot_data)
```

CODE OF PCA:

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
principalComponents = pca.fit_transform(x)
```

CODE FOR RANDOM FOREST WITH EXAMPLE:

```
# explore random forest bootstrap sample size on performance
from numpy import mean
from numpy import std
from numpy import arange
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import RandomForestClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
        X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
random_state=3)
        return X, y

# get a list of models to evaluate
def get_models():
        models = dict()
        # explore ratios from 10% to 100% in 10% increments
        for i in arange(0.1, 1.1, 0.1):
                key = '%.1f' % i
                # set max_samples=None to use 100%
                if i == 1.0:
                        i = None
```

```
                models[key] = RandomForestClassifier(max_samples=i)
        return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
        # define the evaluation procedure
        cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
        # evaluate the model and collect the results
        scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
        return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
        # evaluate the model
        scores = evaluate_model(model, X, y)
        # store the results
        results.append(scores)
        names.append(name)
        # summarize the performance along the way
        print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```