**[Model List]**

| **Numerical** | **Categorical** | **Other** |
| --- | --- | --- |
| 1. Linear Regression | 1. Logistic Regression | 1. Neural Networks |
| 1. Multiple Linear Regression | 1. Classification Tree | 1. PCA   (Dimensionality Reduction) |
| 1. Regularized Linear Regression (L1 Lasso) | 1. Random Forest | 1. K-Means Clustering (Clustering) |
| 1. Regularized Linear Regression (L2 Ridge) | 1. Naive Bayes |  |
| 1. k-Nearest Neighbor(kNN) | 1. k-Nearest Neighbor(kNN) |  |
|  | 1. Support Vector Machines(SVM) | **Total:** 11 Models |

**[Numerical Models]**

**Linear Regression**

**Code**

# Importing necessary libraries

import numpy as np

from sklearn.linear\_model import LinearRegression

import matplotlib.pyplot as plt

# Generating sample data

np.random.seed(0)

X = 2 \* np.random.rand(100, 1)

# Generate 100 random numbers between 0 and 2

y = 4 + 3 \* X + np.random.randn(100, 1)

# Linear relationship with some noise

# Visualizing the data

plt.scatter(X, y, color='blue')

plt.xlabel('X')

plt.ylabel('y')

plt.title('Sample Data')

plt.show()

# Fitting the linear regression model

model = LinearRegression()

model.fit(X, y)

# Printing the coefficients (intercept and slope)

print("Intercept:", model.intercept\_)

print("Slope:", model.coef\_[0])

# Making predictions

X\_new = np.array([[0], [2]])

# New data points to predict

y\_pred = model.predict(X\_new)

# Visualizing the linear regression line

plt.scatter(X, y, color='blue')

plt.plot(X\_new, y\_pred, color='red', linewidth=2)

plt.xlabel('X')

plt.ylabel('y')

plt.title('Linear Regression')

plt.show()

**Multiple Linear Regression**

• Multiple Linear Regression (MLR) is a fundamental and widely used model in statistical learning. It’s considered the cornerstone of many predictive modeling techniques due to its simplicity and interpretability.  
• In MLR, the target variable (y) is assumed to be a linear combination of the input variables (X1, X2, . . . , Xn).

The model prediction (ˆy) can be expressed as:  


where:  
– w0, w1, . . . , wn are the coefficients or weights of the model.  
– X1, X2, . . . , Xn are the input variables.  
• The learning algorithm, Ordinary Least Squares (OLS), aims to find the best combination of weights  
(w) that minimizes the Residual Sum of Squares (RSS), defined as:

A black and white math equation

Description automatically generated

where:  
– N is the number of observations in the training set.  
– yi represents the actual values of the target.  
– ˆyi are the predicted values.  
• The goal is to adjust the weights such that the predicted values (ˆy) are as close as possible to the actual target values (y), thereby minimizing the RSS.

Steps for Model Training and Prediction in Scikit-Learn  
1. Import the estimator class you will use.  
2. Create an instance of the class (instantiate the object). Here, you provide any extra parameters; some of them are model hyperparameters.  
3. Use the fit method of the instance; this step trains the model.  
4. Use the predict method to get the predictions.

**Code:**

# Importing necessary libraries

import numpy as np

from sklearn.linear\_model import LinearRegression

# Generating sample data

np.random.seed(0)

X = np.random.rand(100, 3)

# Generate 100 samples with 3 features

y = 2 + 3 \* X[:,0] + 4 \* X[:,1] + 5 \* X[:,2] + np.random.randn(100)

# Linear relationship with some noise

# Fitting the multiple linear regression model

model = LinearRegression()

model.fit(X, y)

# Printing the coefficients (intercept and slopes)

print("Intercept:", model.intercept\_)

print("Coefficients:", model.coef\_)

# Making predictions (you can replace X\_new with your own data)

X\_new = np.array([[0.2, 0.3, 0.4], [0.5, 0.6, 0.7]])

# New data points to predict

y\_pred = model.predict(X\_new)

# Printing the predictions

print("Predictions:", y\_pred)

**Regularized Linear Regression**

Regularization is a technique used in machine learning to prevent overfitting by imposing a penalty on the  
magnitude of the coefficients of features in a model. The two most common types of regularization are L1  
(Lasso) and L2 (Ridge) regularization.

**Regularization (Lasso):** L1 regularization adds a penalty equal to the absolute value of the magnitude of coefficients. This can lead not only to smaller coefficients but can also result in setting coefficients to zero, effectively selecting a simpler model that may exclude some features entirely. For a regression problem, the objective function with L1 regularization can be formulated as follows:

A black text on a white background

Description automatically generated

Where:

• Loss Function is the loss you are trying to minimize (e.g., Mean Squared Error for regression problems).  
For Mean Squared Error, the loss function is , where *m* is the number of samples,  
y(j) is the actual value and ˆy(j) is the predicted value.  
• wi represents the coefficients or weights of the model.  
• *n* is the number of features.  
• λ is the regularization strength (hyperparameter). It controls the trade-off between fitting the model  
well on the training data and keeping the model weights small to avoid overfitting. Higher values of λ  
increase the regularization strength. where wi are the coefficients, and λ is the regularization strength.  
The regularization strength, λ, controls the trade-off between allowing the model to fit the data while  
keeping the model weights small. Finding a good value of λ is critical and is usually done through  
cross-validation.

**L2 Regularization (Ridge):** L2 regularization adds a penalty equal to the square of the magnitude of coefficients. This discourages large coefficients but does not set them to zero. This can be particularly useful when you suspect many features contribute small but important effects. The mathematical representation is:

A close up of a word

Description automatically generated

**Code**

# Importing necessary libraries

import numpy as np

from sklearn.linear\_model import Ridge

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import mean\_squared\_error

# Generating sample data

np.random.seed(0)

X = np.random.rand(100, 3) # Generate 100 samples with 3 features

y = 2 + 3 \* X[:,0] + 4 \* X[:,1] + 5 \* X[:,2] + np.random.randn(100)

# Linear relationship with some noise

# Splitting the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Fitting the Ridge regression model

alpha = 0.1 # Regularization parameter

model = Ridge(alpha=alpha)

model.fit(X\_train, y\_train)

# Making predictions on the testing set

y\_pred = model.predict(X\_test)

# Calculating Mean Squared Error (MSE)

mse = mean\_squared\_error(y\_test, y\_pred)

print("Mean Squared Error:", mse)

**k-Nearest Neighbors (kNN)**

The kNN algorithm works on a simple principle: it classifies a data point based on how its neighbors are  
classified. In kNN, ’neighbors’ are the data points in the training set that are nearest to the point in question.

Algorithm Steps  
1. Choose the number of k and a distance metric (e.g., Euclidean distance).  
2. Find the k nearest neighbors of the sample that we want to classify.  
3. Assign the class label by majority vote. For regression tasks, assign the average of the k nearest  
neighbors.

**Code**

# Importing necessary libraries

import numpy as np

from sklearn.datasets import make\_classification

from sklearn.model\_selection import train\_test\_split

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy\_score

# Generating synthetic data

X, y = make\_classification(n\_samples=100, n\_features=2, n\_classes=2, n\_clusters\_per\_class=1, random\_state=42)

# Splitting the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Fitting the kNN classifier

k = 5 # Number of neighbors

model = KNeighborsClassifier(n\_neighbors=k)

model.fit(X\_train, y\_train)

# Making predictions on the testing set

y\_pred = model.predict(X\_test)

# Calculating accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

print("Accuracy:", accuracy)

**[Categorical Models]**

**Logistic Regression**

* Logistic regression is a statistical method for predicting the outcome of a binary variable based on one or

more predictor variables. It models the probability that a given input point belongs to a certain category.

Mathematical Explanation

* Logistic Function: The logistic function, or sigmoid function, is central to logistic regression and is defined as:

A number and plus symbol

Description automatically generated with medium confidence

Where:

*z* is the linear combination of input features xi, weighted by coefficients βi

i.e. *z* = β0 + β1x1 + β2x2 +. . . + βnxn

* Estimating Probabilities: The probability that Y = 1, given X, is modeled as:

P (Y = 1|X) = f (β0 + β1x1 + β2x2 + . . . + βnxn)

* Odds Ratio: The odds of the outcome are the ratio of the probability of the event occurring to the probability of the event not occurring:

Odds = = e β0+β1 x1 +β2x2+...+βn xn

* Logit Function: The logit function is the logarithm of the odds ratio and is a linear combination of the input features:

Logit Function = = β0+β1 x1 +β2x2+...+βn xn

**Code**

# Importing necessary libraries

import numpy as np

from sklearn.datasets import make\_classification

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score

# Generating synthetic data

X, y = make\_classification(n\_samples=100, n\_features=2, n\_classes=2, n\_clusters\_per\_class=1, random\_state=42)

# Splitting the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Fitting the logistic regression model

model = LogisticRegression()

model.fit(X\_train, y\_train)

# Making predictions on the testing set

y\_pred = model.predict(X\_test)

# Calculating accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

print("Accuracy:", accuracy)

**Classification Tree**

• Classification trees are a type of decision tree used for separating the dataset into classes based on feature variables.

• They work by iteratively splitting the data based on specific features, creating a tree-like model of decisions.

• Its non-parametric approach allows for flexible modeling without assuming the distribution of input variables.

**Code**

# Importing necessary libraries

import numpy as np

from sklearn.datasets import make\_classification

from sklearn.model\_selection import train\_test\_split

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score, classification\_report

# Generating synthetic data

X, y = make\_classification(n\_samples=100, n\_features=2, n\_classes=2, n\_clusters\_per\_class=1, random\_state=42)

# Splitting the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Fitting the decision tree classifier

model = DecisionTreeClassifier()

model.fit(X\_train, y\_train)

# Making predictions on the testing set

y\_pred = model.predict(X\_test)

# Calculating accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

print("Accuracy:", accuracy)

# Displaying classification report

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred))

**Random Forest**

• Random Forest is an ensemble learning method that constructs multiple decision trees

• By aggregating the predictions of numerous trees, Random Forest achieves higher performance and robustness than individual decision trees.

**Code**

# Importing necessary libraries

import numpy as np

from sklearn.datasets import make\_classification

from sklearn.model\_selection import train\_test\_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy\_score, classification\_report

# Generating synthetic data

X, y = make\_classification(n\_samples=100, n\_features=2, n\_classes=2, n\_clusters\_per\_class=1, random\_state=42)

# Splitting the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Fitting the Random Forest classifier

model = RandomForestClassifier(n\_estimators=100, random\_state=42)

model.fit(X\_train, y\_train)

# Making predictions on the testing set

y\_pred = model.predict(X\_test)

# Calculating accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

print("Accuracy:", accuracy)

# Displaying classification report

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred))

**Naive Bayes**

The core of the Naive Bayes model is Bayes’ theorem, which describes the probability of an event, based on prior knowledge of conditions that might be related to the event. Mathematically, it’s expressed as:

A mathematical equation with a line between the lines

Description automatically generated with medium confidence

Where:

• P (A|B) is the posterior probability of class A given predictor B,

• P (B|A) is the likelihood which is the probability of predictor given class,

• P (A) is the prior probability of class,

• and P (B) is the prior probability of predictor.

**Code**

# Importing necessary libraries

import numpy as np

from sklearn.datasets import make\_classification

from sklearn.model\_selection import train\_test\_split

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import accuracy\_score, classification\_report

# Generating synthetic data

X, y = make\_classification(n\_samples=100, n\_features=2, n\_classes=2, n\_clusters\_per\_class=1, random\_state=42)

# Splitting the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Fitting the Naive Bayes classifier

model = GaussianNB()

model.fit(X\_train, y\_train)

# Making predictions on the testing set

y\_pred = model.predict(X\_test)

# Calculating accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

print("Accuracy:", accuracy)

# Displaying classification report

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred))