[Model List]

Numerical	Categorical	Other
1. Linear Regression	1. Logistic Regression	1. Neural Networks
2. Multiple Linear Regression	2. Classification Tree	2. PCA (Dimensionality Reduction)
3. Regularized Linear Regression (L1 Lasso)	3. Random Forest	3. K-Means Clustering (Clustering)
4. Regularized Linear Regression (L2 Ridge)	4. Naive Bayes	
5. k-Nearest Neighbor(kNN)	5. k-Nearest Neighbor(kNN)	
	6. 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:

$$\hat{y} = w_0 + w_1 X_1 + w_2 X_2 + \ldots + w_n X_n$$

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:

$$RSS = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

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:

$$\text{Objective}_{L1} = \text{Loss Function} + \lambda \sum_{i=1}^{n} |w_i|$$

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 $\frac{1}{m}\sum_{j=1}^{m}(\bar{y}^{(j)} - \hat{y}^{(j)})^{\bar{2}}$, where *m* is the number of samples, $y^{(j)}$ is the actual value and $\hat{}y^{(j)}$ is the predicted value.
• w_i 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:

$$\text{Objective}_{L2} = \text{Loss Function} + \lambda \sum_{i=1}^{n} w_i^2$$

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:

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

Where:

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

$$\text{i.e. } z = \beta_0 + \beta 1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$$

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

$$P(Y = 1|X) = f(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n)$$

- 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:

$$\text{Odds} = \frac{P(Y = 1|X)}{1 - P(Y = 1|X)} = e^{\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n}$$

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

$$\text{Logit Function} = \frac{P(Y = 1|X)}{1 - P(Y = 1|X)} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$$

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:

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

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))
```