Logistic

```
class LogisticRegression:
    def __init__(self, learning_rate=0.01, num_itrtns=10000):
        self.lr = learning_rate
        self.num_itrtns = num_itrtns
        self.wghts = None
        self.bias = None

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def fit(self, X, y):
        num_smpls, num_fts = X.shape
        self.wghts = np.zeros(num_fts)
        self.bias = 0

        # Gradient descent
        for _ in range(self.num_itrtns):
            model = np.dot(X, self.wghts) + self.bias
            predictions = self.sigmoid(model)

            # Gradient calculation
            dw = (1 / num_smpls) * np.dot(X.T, (predictions - y))
            db = (1 / num_smpls) * np.sum(predictions - y)

            # Update weights and bias
            self.wghts -= self.lr * dw
            self.bias -= self.lr * db

    def predict(self, X):
        model = np.dot(X, self.wghts) + self.bias
```

```
        predictions = self.sigmoid(model)
        predictions_cls = [1 if i > 0.5 else 0 for i in predictions]
        return predictions_cls
```

KNN

```
def euclidean_distance(p1, p2):
    return np.sqrt(np.sum((p1 - p2)**2))
def vote(neighbours):
    class_counter = Counter(neighbours)
    return class_counter.most_common(1)[0][0]
def knn(x_train, y_train, test_point, k):
    distances = []
    for i in range(len(x_train)):
        distance = euclidean_distance(test_point, x_train.iloc[i])
        distances.append((distance, y_train.iloc[i]))
    distances = sorted(distances)
    neighbours = np.asarray(distances[:k])
    label = vote(neighbours[:, 1])
    return label
```

SVM

```
class SVM:
    def __init__(self, learning_rate=0.001,n_iters=1000):
        self.lr = learning_rate
        self.n_iters = n_iters
        self.w = None
        self.b = None
```

```
    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.w = np.zeros(n_features)
        self.b = 0

        for _ in range(self.n_iters):
            for idx, x_i in enumerate(X.values):
                condition = y.iloc[idx] * (np.dot(x_i, self.w) + self.b) >= 1
                if condition:
                    self.w -= self.lr * (2 * self.w)
                else:
                    self.w -= self.lr * (2*self.w - np.dot(x_i, y.iloc[idx]))
                    self.b -= self.lr * y.iloc[idx]

    def predict(self, X):
        prediction = np.dot(X, self.w) + self.b
        return np.sign(prediction).astype(int)
```

Linear

```
def fit(X_train,y_train):
    m=0
    b=0
    num=0
    den=0
    for i in range(X_train.shape[0]):
        num = num +((X_train[i] - X_train.mean())*(y_train[i] - y_train.mean()))
        den = den +((X_train[i] - X_train.mean())*(X_train[i] - X_train.mean()))
    m = num/den
    b = y_train.mean() - (m*X_train.mean())
    print(m)
    print(b)
```

```
    return m,b

def predict(m,b,X_test):
    print(X_test)
    y = m*X_test + b
    return y
```

Decision tree

```
def entropy(y):
 class_counts = {}
 for label in y:
 if label in class_counts:
 class_counts[label] += 1
 else:
 class_counts[label] = 1
 entropy = 0
 total_samples = len(y)
 for count in class_counts.values():
 probability = count / total_samples
 entropy -= probability * math.log2(probability)
 return entropy
def information_gain(y, feature):
 total_entropy = entropy(y)

 unique_values = feature.unique()
 weighted_entropies = 0
 for value in unique_values:
 subset_y = y[feature == value]
 weighted_entropies += len(subset_y) / len(y) * entropy(subset_y)
```

```
information_gain= total_entropy - weighted_entropies
return information_gain
class Node:
def __init__(self, feature=None, value=None, entropy=None, information_gain=None, left=None,
right=None):
self.feature = feature
self.value = value
self.entropy = entropy
self.information_gain = information_gain
self.left = left
self.right = right
def build_decision_tree(X, y):
if entropy(y) == 0:
# If all instances have the same class, create a leaf node
return Node(value=y.iloc[0])
if X.empty:
# If no features left, create a leaf node with the majority class
return Node(value=y.value_counts().idxmax())
# Find the best feature to split on
best_feature = None
max_info_gain = -1
for feature_name in X.columns:
current_info_gain = information_gain(y, X[feature_name])
if current_info_gain > max_info_gain:
max_info_gain = current_info_gain
best_feature = feature_name
# Create a node with the best feature
node = Node(feature=best_feature, entropy=entropy(y), information_gain=max_info_gain, value={})
# Recursively build the left and right subtrees
unique_values = X[best_feature].unique()
for value in unique_values:
```

```
 subset_X = X[X[best_feature] == value].drop(columns=[best_feature])
 subset_y = y[X[best_feature] == value]
 child_node = build_decision_tree(subset_X, subset_y)
 if node.value is None:
 node.value = {value: child_node}
 else:
 node.value[value] = child_node
 return node
def predict(node, instance):
 if node.feature is None:
 return node.value
 else:
 value = instance[node.feature]
 if value in node.value:
 return predict(node.value[value], instance)
 else:
 return node.value
Predictions = [predict(tree, instance) for _, instance in X_test.iterrows()]
```

Naïve Bayes

```
from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()
for column in df.select_dtypes(include=['object']).columns:
 df[column] = label_encoder.fit_transform(df[column])
def naive_bayes(X, y):
 class_probs = {}
 feature_probs = {}
 num_samples, num_features = X.shape
 unique_classes = np.unique(y)
 for c in unique_classes:
```

```
 # Calculate class probabilities
 class_probs[c] = np.sum(y == c) / num_samples
 # Calculate feature probabilities for each class
 features_given_class = X[y == c]
 feature_probs[c] = np.sum(features_given_class, axis=0) / np.sum(y == c)
 return class_probs, feature_probs
def predict(X, class_probs, feature_probs):
 predictions = []
 for sample in X:
 class_scores = {}
 for c, class_prob in class_probs.items():
 # Calculate the probability of the sample belonging to each class
 feature_probs_given_class = feature_probs[c]
 log_prob = np.sum(np.log(sample * feature_probs_given_class + (1 - sample) * (1 -
feature_probs_given_class)))
 class_scores[c] = np.log(class_prob) + log_prob
 # Predict the class with the highest probability
 predicted_class = max(class_scores, key=class_scores.get)
 predictions.append(predicted_class)
 return predictions
```

PCA

```
def calculate_mean(data):
 return np.mean(data, axis=0)
def calculate_covariance_matrix(data, mean):
 n_samples = data.shape[0]
 covariance_matrix = np.dot((data - mean).T, (data - mean)) / (n_samples - 1)
 return covariance_matrix
def perform_eigenvalue_decomposition(cov_matrix):
 eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)
```

```
 return eigenvalues, eigenvectors
def sort_eigenvectors(eigenvalues, eigenvectors):
 sorted_indices = np.argsort(eigenvalues)[::-1]
 sorted_eigenvalues = eigenvalues[sorted_indices]
 sorted_eigenvectors = eigenvectors[:, sorted_indices]
 return sorted_eigenvalues, sorted_eigenvectors
def select_principal_components(eigenvectors, num_components):
 selected_components = eigenvectors[:, :num_components]
 return selected_components
def transform_data(data, selected_components):
 transformed_data = np.dot(data, selected_components)
 return transformed_data
def pca(data, num_components):
 mean = calculate_mean(data)
 covariance_matrix = calculate_covariance_matrix(data, mean)
 eigenvalues, eigenvectors = perform_eigenvalue_decomposition(covariance_matrix)
 sorted_eigenvalues, sorted_eigenvectors = sort_eigenvectors(eigenvalues, eigenvectors)

 selected_components = select_principal_components(sorted_eigenvectors, num_components)

 transformed_data = transform_data(data, selected_components)
 return transformed_data
```

RANDOM FOREST

```
class RandomForest:
 def __init__(self, n_estimators=100, max_depth=None):
 self.n_estimators = n_estimators
 self.max_depth = max_depth
 self.trees = []
 def fit(self, X, y):
```

```
for _ in range(self.n_estimators):
tree = DecisionTreeClassifier(max_depth=self.max_depth)
indices = np.random.choice(len(X), len(X), replace=False)
tree.fit(X[indices], y[indices])
self.trees.append(tree)
def predict(self, X):
predictions = np.array([tree.predict(X) for tree in self.trees])
return np.apply_along_axis(lambda x: np.bincount(x).argmax(), axis=0, arr=predictions)
```

K-Means

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

def euclidean_distance(x1, x2):
    return np.sqrt(np.sum((x1 - x2) ** 2))

def initialize_centroids(X, k):
    centroids = X[np.random.choice(X.shape[0], k, replace=False)]
    return centroids

def assign_clusters(X, centroids):
    distances = np.array([np.linalg.norm(X - centroid, axis=1) for centroid in centroids])
    clusters = np.argmin(distances, axis=0)
    return clusters

def update_centroids(X, clusters, k):
    centroids = np.array([X[clusters == i].mean(axis=0) for i in range(k)])
    return centroids
```

```python
def kmeans(X, k, max_iters=100):
    centroids = initialize_centroids(X, k)
    for _ in range(max_iters):
        prev_centroids = centroids.copy()
        clusters = assign_clusters(X, centroids)
        centroids = update_centroids(X, clusters, k)
        if np.all(prev_centroids == centroids):
            break
    return clusters, centroids

# Load the dataset
df = pd.read_csv("student_clustering.csv")
X = df[['cgpa', 'iq']].values



# Choose the optimal number of clusters based on the elbow method
optimal_k = 4

# Run K-means with the optimal K
clusters, centroids = kmeans(X, optimal_k)

# Visualize the clustering results
plt.figure(figsize=(10, 6))
for i in range(optimal_k):
    plt.scatter(X[clusters == i][:, 0], X[clusters == i][:, 1], label=f'Cluster {i}', alpha=0.6, s=100)
plt.scatter(centroids[:, 0], centroids[:, 1], s=300, c='red', label='Centroids')
plt.xlabel('CGPA')
plt.ylabel('IQ')
plt.title('K-means Clustering')
plt.legend()
plt.show()
```

ANN_And

Perceptron:

```
import numpy as np

weight = np.array([0, 0])
bias = 0
learning_rate = 1

def activation(yin):
    if yin > 0:
        return 1
    elif yin == 0:
        return 0
    else:
        return -1

epochs = 100

input_data = np.array([[1, 1], [1, -1], [-1, 1], [-1, -1]])
targets = np.array([1, -1, -1, -1])

for epoch in range(epochs):
    for i in range(len(input_data)):
        yin = bias + np.dot(input_data[i], weight)
        y = activation(yin)
        if y != targets[i]:
            weight[0] += learning_rate * targets[i] * input_data[i][0]
            weight[1] += learning_rate * targets[i] * input_data[i][1]
            bias += learning_rate * targets[i]
```

```
print("Final weight:", weight)
print("Final bias:", bias)

def predict(x1, x2):
    yin = bias + np.dot([x1, x2], weight)
    return activation(yin)

# Test the predict function
print("Input (1, 1): Predicted Output:", predict(1, 1))
print("Input (1, -1): Predicted Output:", predict(1, -1))
print("Input (-1, 1): Predicted Output:", predict(-1, 1))
print("Input (-1, -1): Predicted Output:", predict(-1, -1))
```