ML Experiments

2nd Experiment

```
import numpy as np
import matplotlib.pyplot as plt
# Sample data (you can change these values to your dataset)
# Independent variable (X) and dependent variable (Y)
X = np.array([3, 5, 7, 9, 11, 13, 15, 17, 19, 21])
Y = np.array([4, 8, 6, 12, 14, 16, 18, 20, 22, 24])
# Number of observations
n = len(X)
# Mean of X and Y
mean_X = np.mean(X)
mean_Y = np.mean(Y)
# Calculating the coefficients
# Using the formula: slope (m) = Σ((X - X_mean) * (Y - Y_mean)) / Σ((X - X_mean)^2)
numerator = np.sum((X - mean_X) * (Y - mean_Y))
denominator = np.sum((X - mean_X) ** 2)
m = numerator / denominator
# Intercept (b) = Y_mean - m * X_mean
b = mean_Y - m * mean_X
# Regression line equation: Y = m * X + b
print(f"Slope (m): {m}")
print(f"Intercept (b): {b}")
print(f"Equation of the line: Y = {m:.2f}X + {b:.2f}")
# Predicting values using the linear regression equation
Y_pred = m * X + b
# Plotting the data points and the regression line
plt.scatter(X, Y, color='blue', label='Original Data')
plt.plot(X, Y_pred, color='red', label='Regression Line')
plt.xlabel('X')
plt.ylabel('Y')
```

```
plt.title('Linear Regression')
plt.legend()
plt.show()
```

3rd Experiment

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
a=int(input("Enter hours:"))
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
hours_of_study = np.array([1, 2, 3, 4, 5, 6, 7])
pass_fail = np.array([0, 0, 0, 1, 1, 1, 1])
X = hours_of_study.reshape(-1, 1)
y = pass_fail
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
model = LogisticRegression()
model.fit(X_scaled, y)
predicted_probabilities = model.predict_proba(X_scaled)[:, 1]
predicted_classes = model.predict(X_scaled)
print(f'Intercept: {model.intercept_[0]}')
print(f'Coefficient: {model.coef_[0][0]}')
hours_to_predict = np.array([[a]])
hours_to_predict_scaled = scaler.transform(hours_to_predict)
predicted_probability = model.predict_proba(hours_to_predict_scaled)[0][1]
print(f'Probability of passing if studied for {a} hours: {predicted_probability:.3f}')
plt.figure(figsize=(7.5, 4.5))
plt.scatter(hours_of_study, pass_fail, color='blue', label='Data')
x_values = np.linspace(0, 8, 100)
x_values_scaled = scaler.transform(x_values.reshape(-1, 1))
```

```
y_values = sigmoid(model.intercept_[0] + model.coef_[0][0] * x_values_scaled.flatten())
plt.plot(x_values, y_values, color='red', label='Logistic Regression Curve')
plt.xlabel('Hours of Study')
plt.ylabel('Probability of Passing')
plt.title('Logistic Regression - Hours of Study vs Probability of Passing')
plt.legend()
plt.grid(True)
plt.show()
accuracy = accuracy_score(y, predicted_classes)
print(f'Accuracy on training data: {accuracy:.2f}')
```

4th Experiment

```
# Import necessary libraries
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
from sklearn import tree
# Load Iris dataset (replace this with any dataset you want)
data = load_iris()
X = data.data  # Features
y = data.target  # Target (Labels)
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
# Initialize the decision tree classifier
clf = DecisionTreeClassifier(criterion='gini', max_depth=3, random_state=42)
# Train the model
clf.fit(X_train, y_train)
# Make predictions on the test set
y_pred = clf.predict(X_test)
```

```
# Calculate accuracy
train_accuracy = accuracy_score(y_train, clf.predict(X_train))
test_accuracy = accuracy_score(y_test, y_pred)
# Output the results
print(f"Training Accuracy: {train_accuracy * 100:.2f}%")
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
# Plot the decision tree
plt.figure(figsize=(12,8))
tree.plot_tree(clf, feature_names=data.feature_names, class_names=data.target_names, filled=True, rounded=True)
plt.title("Decision Tree")
plt.show()
```

5th Experiment

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import (confusion_matrix, ConfusionMatrixDisplay,
                             accuracy_score, precision_score, recall_score,
                             f1_score)

# Load the Iris dataset
data = load_iris()
X = data.data  # Features
y = data.target  # Target (Labels)
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
# Initialize the decision tree classifier
clf = DecisionTreeClassifier(criterion='gini', max_depth=3, random_state=42)
# Train the model
clf.fit(X_train, y_train)
```

```
# Make predictions on the test set
y_pred = clf.predict(X_test)
# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Test Accuracy: {accuracy * 100:.2f}%")
# Generate the confusion matrix
cm = confusion_matrix(y_test, y_pred)
# Display the confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=data.target_names)
disp.plot(cmap=plt.cm.Blues)
plt.title("Confusion Matrix")
plt.show()
# Calculate precision, recall, and F1 score for each class
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')
# Calculate Type I and Type II errors
# Type I Error (False Positive Rate)
TP = cm.diagonal()  # True Positives
FP = cm.sum(axis=0) - TP  # False Positives
FN = cm.sum(axis=1) - TP  # False Negatives
# Specificity
specificity = TN = np.zeros(len(data.target_names))
for i in range(len(data.target_names)):
    TN[i] = cm.sum() - (FP[i] + FN[i] + TP[i])
type_1_error = FP / (FP + TP)  # False Positive Rate for each class
type_2_error = FN / (FN + TP)  # False Negative Rate for each class
# Display metrics
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1 Score: {f1:.2f}")
```

```
print("Specificity for each class:")
for i, class_name in enumerate(data.target_names):
    print(f"{class_name}: {specificity[i]:.2f}")
print("Type I Error for each class:")
for i, class_name in enumerate(data.target_names):
    print(f"{class_name}: {type_1_error[i]:.2f}")
print("Type II Error for each class:")
for i, class_name in enumerate(data.target_names):
    print(f"{class_name}: {type_2_error[i]:.2f}")
```

6th Experiment

```
from sklearn import datasets
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics
# Load the iris dataset
flower_dataset = datasets.load_iris()
# Print basic information about the dataset
print(flower_dataset.target_names)
print(flower_dataset.feature_names)
print(flower_dataset.data[0:5])
print(flower_dataset.target)
# Create a DataFrame from the iris data
flower_df = pd.DataFrame({
    'sepal_length': flower_dataset.data[:, 0],
    'sepal_width': flower_dataset.data[:, 1],
    'petal_length': flower_dataset.data[:, 2],
    'petal_width': flower_dataset.data[:, 3],
    'species_label': flower_dataset.target
})
# Display the first few rows of the DataFrame
print(flower_df.head())
```

```
# Split the data into features and labels
input_features = flower_df[['sepal_length', 'sepal_width', 'petal_length', 'petal_width']]  # Features
output_labels = flower_df['species_label']  # Labels
# Split the dataset into training and testing sets
input_train, input_test, output_train, output_test = train_test_split(
    input_features, output_labels, test_size=0.3, random_state=42
)
# Initialize the RandomForestClassifier
forest_classifier = RandomForestClassifier(n_estimators=100)
# Fit the classifier on the training data
forest_classifier.fit(input_train, output_train)
# Make predictions on the test data
predicted_labels = forest_classifier.predict(input_test)
# Calculate and print the accuracy of the model
print("Classification Accuracy: ", metrics.accuracy_score(output_test, predicted_labels))
# Make a prediction on new data
new_sample = pd.DataFrame([[5, 2.5, 3.5, 1]], columns=['sepal_length', 'sepal_width', 'petal_length',
'petal_width'])
species_prediction = forest_classifier.predict(new_sample)
# Output the predicted species
if species_prediction[0] == 0:
    print('Predicted Species: Setosa')
elif species_prediction[0] == 1:
    print('Predicted Species: Versicolor')
else:
    print('Predicted Species: Virginica')
```

7th Experiment

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
```

```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
# Generate synthetic data
X, y = datasets.make_classification(n_samples=100, n_features=2, n_informative=2,
                                    n_redundant=0, n_clusters_per_class=1, random_state=42)
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
# Initialize and fit the SVM model
svm_model = SVC(kernel='linear', C=1.0, random_state=42)
svm_model.fit(X_train, y_train)
# Make predictions and calculate accuracy
y_pred = svm_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
def plot_decision_boundary(X, y, model):
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 500), np.linspace(y_min, y_max, 500))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.figure(figsize=(10, 6))
    plt.contourf(xx, yy, Z, alpha=0.3, cmap=plt.cm.RdYlBu)
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', cmap=plt.cm.RdYlBu)
    plt.title("SVM Decision Boundary")
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.show()
# Plot the decision boundary
plot_decision_boundary(X_test, y_test, svm_model)
```

8th Experiment

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import SpectralClustering
from sklearn.metrics.pairwise import pairwise_distances
import networkx as nx
# Generate synthetic data
X, y = datasets.make_classification(n_samples=100, n_features=2, n_informative=2,
                                    n_redundant=0, n_clusters_per_class=1, random_state=42)
# Apply Spectral Clustering
n_clusters = len(np.unique(y))  # Number of clusters based on the number of classes
spectral = SpectralClustering(n_clusters=n_clusters, affinity='nearest_neighbors', random_state=42)
labels = spectral.fit_predict(X)
def plot_clusters(X, labels):
    plt.figure(figsize=(10, 6))
    scatter = plt.scatter(X[:, 0], X[:, 1], c=labels, edgecolors='k', cmap=plt.cm.RdYlBu)
    plt.title("Clusters with Spectral Clustering")
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.colorbar(scatter)
    plt.show()
plot_clusters(X, labels)
# Create a graph from pairwise distances
distances = pairwise_distances(X)
G = nx.Graph()
# Add edges based on distance
for i in range(len(X)):
    for j in range(i + 1, len(X)):
        G.add_edge(i, j, weight=distances[i, j])
# Compute the MST
mst = nx.minimum_spanning_tree(G)
```

```
# Convert MST to a simple graph for visualization
mst = nx.Graph(mst)
def plot_mst(X, mst):
    pos = {i: (X[i, 0], X[i, 1]) for i in range(len(X))}
    plt.figure(figsize=(10, 6))
    nx.draw(mst, pos, with_labels=True, node_color='lightblue', edge_color='gray',
            node_size=100, font_size=12, font_weight='bold')
    plt.title("Minimum Spanning Tree")
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.show()
plot_mst(X, mst)
```

9th Experiment

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.cluster import DBSCAN
from sklearn import metrics
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler
from sklearn import datasets
# Load data in X
X, y_true = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)
# Standardize the dataset
X = StandardScaler().fit_transform(X)
# Create DBSCAN model
db = DBSCAN(eps=0.3, min_samples=10).fit(X)
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
labels = db.labels_
# Number of clusters in labels, ignoring noise if present
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
```

```
# Plot result
# Black removed and is used for noise instead
unique_labels = set(labels)
colors = ['y', 'b', 'g', 'r']
print(colors)
for k, col in zip(unique_labels, colors):
    if k == -1:
        # Black used for noise
        col = 'k'
    class_member_mask = (labels == k)
    xy = X[class_member_mask & core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=col,
             markeredgecolor='k', markersize=6)
    xy = X[class_member_mask & ~core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=col,
             markeredgecolor='k', markersize=6)
plt.title('Number of clusters: %d' % n_clusters_)
plt.show()
sc = metrics.silhouette_score(X, labels)
print("Silhouette Coefficient: %0.2f" % sc)
ari = metrics.adjusted_rand_score(y_true, labels)
print("Adjusted Rand Index: %0.2f" % ari)
```

10th Experiment

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_iris
# Load dataset
data = load_iris()
X = data.data
y = data.target
```

```
feature_names = data.feature_names
# Standardize the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
# Apply PCA
pca = PCA(n_components=4)  # PCA with all components
X_pca = pca.fit_transform(X_scaled)
# Plot 2D projection
plt.figure(figsize=(12, 5))
# Scatter plot of first two principal components
plt.subplot(1, 2, 1)
scatter = plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap='viridis')
plt.colorbar(scatter, label='Target')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('2D PCA Projection')
# Bar chart of explained variance ratio
plt.subplot(1, 2, 2)
explained_variance_ratio = pca.explained_variance_ratio_
plt.bar(range(1, len(explained_variance_ratio) + 1), explained_variance_ratio, alpha=0.7)
plt.xlabel('Principal Component')
plt.ylabel('Explained Variance Ratio')
plt.title('Explained Variance Ratio')
plt.tight_layout()
plt.show()
```