ML

ML Experiments

Experiment - 1 Data Preprocessing

Code

```
import numpy as np
import pandas as pd
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Importing the dataset
data_set = pd.read_csv('Dataset.csv', delimiter=',')

# Ensuring all values are numeric and handling incorrect formatting
data_set = data_set.apply(pd.to_numeric, errors='coerce')

# Extracting independent and dependent variables
x = data_set.iloc[:, 1:].values  # Excluding the 'User' column
y = data_set.iloc[:, 0].values  # Keeping 'User' as the dependent variable

# Handling missing data (Replacing missing data with the mean value)
imputer = SimpleImputer(missing_values=np.nan, strategy='mean')
imputer.fit(x)
x = imputer.transform(x)

# Splitting the dataset into the Training set and Test set
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=0)

# Feature Scaling
sc = StandardScaler()
x_train = sc.fit_transform(x_train)
x_test = sc.transform(x_test)

# Displaying preprocessed data
print("Training data after preprocessing:")
print(x_train[:2])  # Display first 5 rows of processed training data
print("Test data after preprocessing:")
print(x_test[:2])  # Display first 5 rows of processed test data
```

Output

```
PS V:\Deeptanshu Lal\PROJECTS\ML\exp-1> python .\exp-1.py
Training data after preprocessing:
[[-8.26053994e-01 -1.07505153e+00 -7.62096340e-01 -5.24596609e-01
   2.49076318e-01  1.62362277e+00  1.15848016e+00  1.41007834e+00
   1.38276733e+00 -1.02022922e+00 -7.77474412e-01 -4.77520229e-01
   3.23371143e-01  1.75644686e+00  1.62827432e+00 -4.33408312e-01
  -2.65508399e-01 -1.30408487e-01 -1.95430238e-01  5.31193506e-02
  -2.15603792e-01 -6.29084361e-01 -6.04599981e-01 -6.93245573e-01
  -1.01835375e-11]
 [ 1.31358641e+00  2.28498220e-01 -2.51199243e-01 -2.34719881e-01
  -8.11818626e-01 -1.18899523e+00 -1.43257430e+00 -1.09938813e+00
  -1.32370769e+00 -9.43534790e-01 -6.61477890e-01 -1.02414282e+00
  -9.57272685e-01 -8.98025222e-01  1.62827432e+00 -2.59947081e-01
  -6.92394926e-02  6.95018481e-02 -1.37026766e-01 -1.79607780e-01
   4.38227393e+00  1.93072829e+00  1.57456141e+00  2.91627801e+00
  -1.01835375e-11]]
Test data after preprocessing:
[[-1.73450467e+00 -1.63170250e+00 -8.09992942e-01 -1.04332339e+00
  -1.15548882e+00 -1.14237725e+00 -1.33346293e+00 -9.54958403e-01
  -3.10628240e-01 -7.68877464e-02  5.63735379e-01  6.71179414e-01
   2.02853779e+00  1.75644686e+00  1.62827432e+00 -3.60372004e-01
  -1.72538917e-01 -2.51925213e-02 -3.69065282e-02 -8.36230757e-01
  -1.03975169e+00 -1.09109933e+00 -1.15504559e+00 -1.32025202e+00
  -1.01835375e-11]
 [-8.49494949e-02 -4.83169480e-01 -6.82268668e-01 -8.60243347e-01
  -9.53769358e-01  1.62362277e+00 -6.62532338e-02  5.79607425e-01
   1.38276733e+00  6.82387097e-01  1.77444908e+00 -6.55727705e-02
  -5.61051390e-01 -5.52063220e-01 -5.20259412e-01  1.14363996e-01
  -2.55178456e-01 -2.88232436e-01 -3.62297301e-01 -4.04023228e-01
  -4.75861022e-01 -6.85275370e-01 -4.68873666e-01 -1.67915849e-01
  -1.01835375e-11]]
```

Experiment - 2 PCA

Code

```
import numpy as np
import pandas as pd
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import fetch_openml
import matplotlib.pyplot as plt
import seaborn as sns

# For 3D plotting
from mpl_toolkits.mplot3d import Axes3D

# Fetch the MNIST dataset from openml
mnist = fetch_openml('mnist_784')
df_mnist = pd.DataFrame(data=mnist.data)
df_mnist['target'] = mnist.target

# Standardize the data
features_mnist = df_mnist.columns[:-1]
x_mnist = df_mnist.loc[:, features_mnist].values
y_mnist = df_mnist.loc[:, ['target']].values
x_mnist = StandardScaler().fit_transform(x_mnist)

# Apply PCA
pca_mnist = PCA(n_components=3)
principal_components_mnist = pca_mnist.fit_transform(x_mnist)
principal_df_mnist = pd.DataFrame(data=principal_components_mnist, columns=['PC1', 'PC2', 'PC3'])
final_df_mnist = pd.concat([principal_df_mnist, df_mnist[['target']]], axis=1)

# Check Components
print("PCA Components (MNIST):\n", pca_mnist.components_)
print("Explained Variance Ratio (MNIST):\n", pca_mnist.explained_variance_ratio_)

# Visualize and Save the Principal Components
fig_mnist = plt.figure(figsize=(8, 6))
ax_mnist = fig_mnist.add_subplot(111, projection='3d')
ax_mnist.set_title('3D PCA Plot (MNIST)')
targets_mnist = np.unique(df_mnist['target'])
colors_mnist = plt.cm.rainbow(np.linspace(0, 1, len(targets_mnist)))

for target, color in zip(targets_mnist, colors_mnist):
    indices_mnist = final_df_mnist['target'] == target
    ax_mnist.scatter(final_df_mnist.loc[indices_mnist, 'PC1'],
                     final_df_mnist.loc[indices_mnist, 'PC2'],
                     final_df_mnist.loc[indices_mnist, 'PC3'],
                     c=[color],
                     s=10,
                     label=target)
ax_mnist.set_xlabel('Principal Component 1')
ax_mnist.set_ylabel('Principal Component 2')
ax_mnist.set_zlabel('Principal Component 3')
ax_mnist.legend(targets_mnist)

# Save the plot as an image file
plt.savefig('pca_mnist_plot.png')
plt.show()

# Calculate Variance Ratio
print("Explained Variance Ratio (MNIST):", pca_mnist.explained_variance_ratio_)
```

Output

Experiment - 3 Linear Regression: BMI vs Life Expectancy

Code

```
# Importing necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# Load the dataset (replace with the correct path to your file)
path = "Life Expectancy Data.csv"
data = pd.read_csv(path)

# Selecting relevant columns: 'BMI' (independent variable) and 'Life expectancy' (dependent variabl
data = data[['BMI', 'Life expectancy']]
data = data.dropna()  # Drop rows with missing values

# Features (X) and target (y)
X = data['BMI'].values.reshape(-1, 1)  # Reshape for sklearn compatibility
y = data['Life expectancy'].values

# Splitting 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)

# Creating and training the linear regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Making predictions on the test set
y_pred = model.predict(X_test)

# Calculating root mean squared error
rmse = mean_squared_error(y_test, y_pred)**0.5
print(f"Root Mean Squared Error: {rmse:.2f} years")

# Plotting the results
plt.figure(figsize=(10, 6))
plt.scatter(X, y, color='blue', label='Data Points')  # Scatter plot of data points
plt.plot(X_test, model.predict(X_test), color='red', label='Regression Line')  # Regression line
plt.title('Linear Regression: BMI vs Life Expectancy')
plt.xlabel('BMI')
plt.ylabel('Life Expectancy')
plt.legend()
plt.savefig("linear_regression_plot.png")  # Save the plot as an image file
plt.show()
```

Output

Experiment - 4 Toxic Comment Classification using Logistic Regression

Code

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matr

# Load dataset
data = pd.read_csv('Toxic_Comments.csv')
X = data['Comment'].values  # Text comments
y = data['Label'].values   # Target variable (0: Non-toxic, 1: Toxic)

# Split dataset into training (80%) and testing (20%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Convert text data into numerical format using TF-IDF vectorization
vectorizer = TfidfVectorizer()
X_train = vectorizer.fit_transform(X_train)
X_test = vectorizer.transform(X_test)

# Train logistic regression model
model = LogisticRegression()
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Calculate evaluation metrics
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)


# Print evaluation metrics
print(f'Accuracy: {accuracy * 100:.2f}%')
print(f'Precision: {precision:.2f}')
print(f'Recall: {recall:.2f}')
print(f'F1 Score: {f1:.2f}')
print('\nConfusion Matrix:\n', conf_matrix)
print('\nClassification Report:\n', class_report)
```

Output

```
PS V:\Deeptanshu Lal\PROJECTS\ML\exp-4> python .\exp-4.py
Accuracy: 100.00%
Precision: 1.00
Recall: 1.00
F1 Score: 1.00

Confusion Matrix:
 [[47  0]
 [ 0 53]]

Classification Report:
               precision    recall  f1-score   support

           0       1.00      1.00      1.00        47
           1       1.00      1.00      1.00        53

    accuracy                           1.00       100
   macro avg       1.00      1.00      1.00       100
weighted avg       1.00      1.00      1.00       100
```

Experiment - 5 McCulloch-Pitts Neural Network Model

Code

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

class McCullochPittsNeuron:
    def __init__(self, weights, threshold):
        """
        Initialize a McCulloch-Pitts neuron

        Parameters:
        weights (list): The weights for each input
        threshold (float): The threshold for activation
        """
        self.weights = np.array(weights)
        self.threshold = threshold

    def activate(self, inputs):
        """
        Calculate the output of the neuron given the inputs

        Parameters:
        inputs (list): The input values

        Returns:
        int: 1 if the weighted sum is greater than or equal to threshold, 0 otherwise
        """
        # Calculate the weighted sum
        weighted_sum = np.dot(inputs, self.weights)

        # Apply the threshold activation function
        return 1 if weighted_sum >= self.threshold else 0

    def test_logic_gate(self, inputs_matrix):
        """
        Test the neuron on multiple input combinations

        Parameters:
        inputs_matrix (list of lists): Matrix containing input combinations

        Returns:
        list: The outputs for each input combination
        """
        outputs = []
        for inputs in inputs_matrix:
            output = self.activate(inputs)
            outputs.append(output)
            print(f"Inputs: {inputs}, Output: {output}")
        return outputs

def plot_logic_gate(gate_name, inputs, outputs):
    """
    Plot the truth table of a logic gate

    Parameters:
    gate_name (str): The name of the logic gate
    inputs (list of lists): Matrix containing input combinations
    outputs (list): The outputs for each input combination
    """
    plt.figure(figsize=(8, 6))

    # For 2 input gates
    if len(inputs[0]) == 2:
        x1 = [row[0] for row in inputs]
        x2 = [row[1] for row in inputs]
        plt.scatter(x1, x2, c=outputs, cmap='coolwarm', s=200, alpha=0.8)

        for i, (x1_val, x2_val) in enumerate(zip(x1, x2)):
            plt.annotate(f"Output: {outputs[i]}",
                        (x1_val, x2_val),
                        xytext=(10, 5),
                        textcoords='offset points')

        plt.xlabel('Input 1')
        plt.ylabel('Input 2')
        plt.grid(True)
        plt.title(f'{gate_name} Gate')
        plt.savefig(f"{gate_name.lower()}_gate.png")

    # For single input gates (like NOT)
    elif len(inputs[0]) == 1:
        x = [row[0] for row in inputs]
        plt.scatter(x, outputs, c=outputs, cmap='coolwarm', s=200, alpha=0.8)

        for i, x_val in enumerate(x):
            plt.annotate(f"Input: {x_val}, Output: {outputs[i]}",
                        (x_val, outputs[i]),
                        xytext=(10, 5),
                        textcoords='offset points')

        plt.xlabel('Input')
        plt.ylabel('Output')
        plt.grid(True)
        plt.title(f'{gate_name} Gate')
        plt.savefig(f"{gate_name.lower()}_gate.png")

def main():
    # Define input combinations for 2-input logic gates
    inputs_2bit = [
        [0, 0],
        [0, 1],
        [1, 0],
        [1, 1]
    ]

    print("\n===== AND Gate =====")
    # AND gate: both inputs must be 1 to get output 1
    and_neuron = McCullochPittsNeuron(weights=[1, 1], threshold=2)
    and_outputs = and_neuron.test_logic_gate(inputs_2bit)
    plot_logic_gate("AND", inputs_2bit, and_outputs)

    print("\n===== OR Gate =====")
    # OR gate: at least one input must be 1 to get output 1
    or_neuron = McCullochPittsNeuron(weights=[1, 1], threshold=1)
    or_outputs = or_neuron.test_logic_gate(inputs_2bit)
    plot_logic_gate("OR", inputs_2bit, or_outputs)

    print("\n===== NAND Gate =====")
    # NAND gate: only when both inputs are 1, output is 0
    nand_neuron = McCullochPittsNeuron(weights=[-1, -1], threshold=-1)
    nand_outputs = nand_neuron.test_logic_gate(inputs_2bit)
    plot_logic_gate("NAND", inputs_2bit, nand_outputs)

    print("\n===== NOR Gate =====")
    # NOR gate: only when both inputs are 0, output is 1
    nor_neuron = McCullochPittsNeuron(weights=[-1, -1], threshold=0)
    nor_outputs = nor_neuron.test_logic_gate(inputs_2bit)
    plot_logic_gate("NOR", inputs_2bit, nor_outputs)

    # Define input combinations for NOT gate
    inputs_1bit = [
        [0],
        [1]
    ]

    print("\n===== NOT Gate =====")
    # NOT gate: invert the input
    not_neuron = McCullochPittsNeuron(weights=[-1], threshold=0)
    not_outputs = not_neuron.test_logic_gate(inputs_1bit)
    plot_logic_gate("NOT", inputs_1bit, not_outputs)

    # XOR gate cannot be implemented with a single McCulloch-Pitts neuron
    # It requires a network of neurons
    print("\n===== XOR Gate =====")
    print("XOR cannot be implemented with a single McCulloch-Pitts neuron.")
    print("It requires a network of neurons.")

    plt.show()

if __name__ == "__main__":
    main()
```

Output

```
===== AND Gate =====
Inputs: [0, 0], Output: 0
Inputs: [0, 1], Output: 0
Inputs: [1, 0], Output: 0
Inputs: [1, 1], Output: 1

===== OR Gate =====
Inputs: [0, 0], Output: 0
Inputs: [0, 1], Output: 1
Inputs: [1, 0], Output: 1
Inputs: [1, 1], Output: 1

===== NAND Gate =====
Inputs: [0, 0], Output: 1
Inputs: [0, 1], Output: 1
Inputs: [1, 0], Output: 1
Inputs: [1, 1], Output: 0

===== NOR Gate =====
Inputs: [0, 0], Output: 1
Inputs: [0, 1], Output: 0
Inputs: [1, 0], Output: 0
Inputs: [1, 1], Output: 0

===== NOT Gate =====
Inputs: [0], Output: 1
Inputs: [1], Output: 0

===== XOR Gate =====
XOR cannot be implemented with a single McCulloch-Pitts neuron.
It requires a network of neurons.
```

Visual Outputs