Program 1: Outfit Recommender Using Git for Version Control

AIM

To build a simple Outfit Recommender System in Python that suggests outfits based on color and occasion, while learning to use Git for version control and managing the project in Google Colab with GitHub.

Procedure

Step 1: Create GitHub Repository

1. Log into GitHub.
2. Create a new repository → Name: outfit-recommender.
3. Add a description → 'A simple outfit recommendation system using Python.'
4. Initialize with a README file.
5. Copy the repo URL.

Final Code

```
# Step 1: Clone repo
!git clone https://github.com/your-username/outfit-recommender.git
%cd outfit-recommender
# Step 2: Create dataset
!mkdir -p data
with open("data/outfits.csv", "w") as f:
   f.write("outfit,color,occasion\n")
   f.write("Casual Shirt and Jeans,Blue,Casual\n")
   f.write("Black Suit,Black,Formal\n")
   f.write("Red Dress,Red,Party\n")
   f.write("Sports Tracksuit,Grey,Sports\n")
!cat data/outfits.csv
# Step 3: Create recommender class
!mkdir -p recommender
with open("recommender/outfit_recommender.py", "w") as f:
   f.write("""
import pandas as pd
class OutfitRecommender:
   def __init__(self, csv_path):
      self.data = pd.read_csv(csv_path)

   def recommend(self, color=None, occasion=None):
      result = self.data
      if color:
         result = result[result['color'].str.lower() == color.lower()]
      if occasion:
         result = result[result['occasion'].str.lower() == occasion.lower()]
      return result['outfit'].tolist()
""")
# Step 4: Create main program
with open("main.py", "w") as f:
   f.write("""
from recommender.outfit_recommender import OutfitRecommender
def main():
   recommender = OutfitRecommender("data/outfits.csv")
   print("Recommendations for color='Red', occasion='Party':")
   print(recommender.recommend(color='Red', occasion='Party'))
if __name__ == "__main__":
   main()
""")
# Step 5: Run program
!python main.py
# Step 6: Git commands
!git config --global user.email "your-email@example.com"
!git config --global user.name "your-username"
!git add .
!git commit -m "Initial recommender version"
!git push origin main
```

PROGRAM 2-DVC with Git – Data Version Control

AIM

To implement Data Version Control (DVC) with Git for tracking changes in a dataset (Salary_Data.csv) by creating multiple modified versions and managing them efficiently using commits and DVC tracking.

Procedure

1. Initial Setup:
   - Install Python (≥3.8), Git, and DVC (pip install dvc).
   - Create a folder DVC and place Salary_Data.csv inside it.
2. Initialize Git and DVC:
   ```
   git init
   pip install dvc
   python -m dvc init
   ```
3. Track the Dataset:
   ```
   python -m dvc add Salary_Data.csv
   git add Salary_Data.csv.dvc .gitignore
   git commit -m "Track the dataset with DVC"
   ```
4. Modify Dataset – Version 1:
   - Create alterv1.py
   - Run script → Add +1 year to YearsExperience.
   - Track with DVC and commit.
5. Modify Dataset – Version 2:
   - Create alterv2.py
   - Run script → Add +2 years to YearsExperience.
   - Track with DVC and commit.
6. Modify Dataset – Version 3:
   - Create alterv3.py
   - Run script → Add +3 years to YearsExperience.
   - Track with DVC and commit.
7. Check Versions:
   - Use git log --oneline to see commits.
   - Checkout older versions with git checkout <commit_id>.
   - Use dvc pull to restore dataset from DVC storage.

Code

- alterv1.py

```
import pandas as pd
data = pd.read_csv("Salary_Data.csv")
data["YearsExperience"] += 1
data.to_csv("Salary_Data.csv", index=False)
```

- alterv2.py

```
import pandas as pd
data = pd.read_csv("Salary_Data.csv")
data["YearsExperience"] += 2
data.to_csv("Salary_Data.csv", index=False)
```

- alterv3.py

```
import pandas as pd
data = pd.read_csv("Salary_Data.csv")
data["YearsExperience"] += 3
data.to_csv("Salary_Data.csv", index=False)
```

Program 3: Object-Oriented Programming in Machine Learning Applications

Aim

To understand and implement the role of **Object-Oriented Programming (OOP)** concepts in building modular machine learning code by applying them to a classification problem.

Procedure

1. Import the required libraries such as `sklearn` for dataset handling, preprocessing, model building, and evaluation.
2. Create a **DataLoader** class to load and split the dataset into training and testing sets.
3. Create a **Preprocessor** class to scale the features using `StandardScaler`.
4. Create an **MLModel** class to define and train the Decision Tree Classifier.
5. Create an **Evaluator** class to evaluate the model predictions using a classification report.
6. Create a main class **MLApplication** that integrates all the steps and executes the pipeline.
7. Finally, run the application and display the results.

Code

```
# Step 1: Import Required Libraries
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report
# Step 2: Data Loader Class
class DataLoader:
    def __init__(self):
        self.X, self.y = load_iris(return_X_y=True)
    def split(self, test_size=0.3, random_state=42):
        return train_test_split(self.X, self.y,
test_size=test_size,
random_state=random_state)
# Step 3: Preprocessor Class
class Preprocessor:
    def __init__(self):
        self.scaler = StandardScaler()
    def fit_transform(self, X_train):
        return self.scaler.fit_transform(X_train)
    def transform(self, X_test):
        return self.scaler.transform(X_test)
# Step 4: ML Model Class
class MLModel:
    def __init__(self):
        self.model = DecisionTreeClassifier()
    def train(self, X_train, y_train):
        self.model.fit(X_train, y_train)
    def predict(self, X_test):
        return self.model.predict(X_test)
# Step 5: Evaluator Class
class Evaluator:
    def __init__(self, y_true, y_pred):
        self.y_true = y_true
        self.y_pred = y_pred
    def report(self):
        print("Classification Report:\n")
        print(classification_report(self.y_true,
self.y_pred))
# Step 6: Main ML Application Class
class MLApplication:
    def __init__(self):
        self.loader = DataLoader()
        self.preprocessor = Preprocessor()
        self.model = MLModel()
    def run(self):
        # Load and split data
        X_train, X_test, y_train, y_test =
self.loader.split()
        # Preprocess data
        X_train_scaled =
self.preprocessor.fit_transform(X_train)
        X_test_scaled =
self.preprocessor.transform(X_test)
      # Train model
        self.model.train(X_train_scaled, y_train)
        y_pred = self.model.predict(X_test_scaled)
        # Evaluate
        evaluator = Evaluator(y_test, y_pred)
        evaluator.report()
# Step 7: Execute the Pipeline
app = MLApplication()
app.run()
```