

ASSIGNMENT-5.4

A.Dhanalaxmi

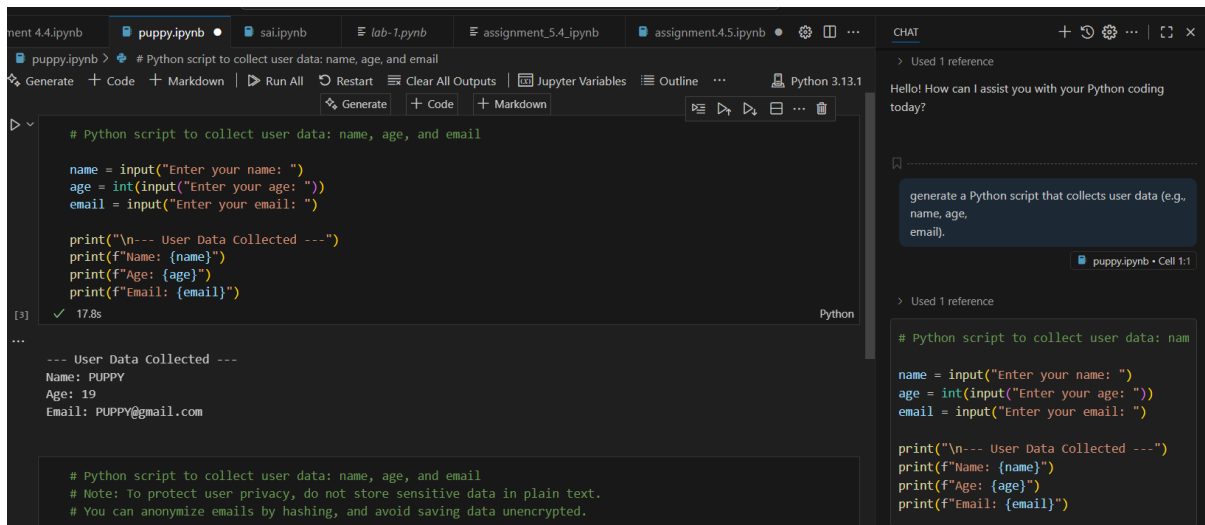
2403A51269

TASK-1:

PROMPT:

generate a Python script that collects user data (e.g., name, age, email)

CODE:



```
# Python script to collect user data: name, age, and email

name = input("Enter your name: ")
age = int(input("Enter your age: "))
email = input("Enter your email: ")

print("\n--- User Data Collected ---")
print(f"Name: {name}")
print(f"Age: {age}")
print(f"Email: {email}")
```

```
--- User Data Collected ---
Name: PUPPY
Age: 19
Email: PUPPY@gmail.com
```

```
# Python script to collect user data: name, age, and email
# Note: To protect user privacy, do not store sensitive data in plain text.
# You can anonymize emails by hashing, and avoid saving data unencrypted.
```

```
# Python script to collect user data: name, age, and email

name = input("Enter your name: ")
age = int(input("Enter your age: "))
email = input("Enter your email: ")

print("\n--- User Data Collected ---")
print(f"Name: {name}")
print(f"Age: {age}")
print(f"Email: {email}")
```

Prompt:

add comments on how to anonymize or protect this data of a script with inline Copilot-suggested code and comments explaining how to safeguard or anonymize user information (e.g., hashing emails, not storing data unencrypted)

Code:

The screenshot shows a Jupyter Notebook environment with a dark theme. The main editor displays a Python script for collecting user data (name, age, email) and hashing the email using SHA256. The script includes comments about privacy and security. The output of the script is visible at the bottom, showing the collected data and the hashed email. On the right side, there is a 'CHAT' sidebar with a message from 'puppy.ipynb' suggesting ways to anonymize or protect data, such as using pseudonyms or hashing emails. The sidebar also shows a 'Used 1 reference' section with a snippet of the script's code.

```
# Python script to collect user data: name, age, and email
# Note: To protect user privacy, do not store sensitive data in plain text.
# You can anonymize emails by hashing, and avoid saving data unencrypted.

import hashlib

name = input("Enter your name: ")
age = int(input("Enter your age: "))
email = input("Enter your email: ")

# Hash the email to anonymize it before storing or processing
hashed_email = hashlib.sha256(email.encode()).hexdigest()

print("\n--- User Data Collected ---")
print(f"Name: {name}") # Consider using pseudonyms or initials for more privacy
print(f"Age: {age}")
print(f"Email (hashed): {hashed_email}") # Only display or store the hashed version

# Tip: Never store raw emails or other sensitive info in logs or files without encryption.
# For more security, use secure databases and restrict access to sensitive data.p
```

Output:

```
--- User Data Collected ---
Name: puppy
Age: 19
Email (hashed): 89da8e528b7280e946c2d69af24510c99a1f83dfbfbdede7da56511a2b8e8556
```

CHAT sidebar:

add comments on how to anonymize or protect this data.

Expected Output #1:

- A script with inline Copilot-suggested code and comments explaining how to safeguard or anonymize user information (e.g., hashing emails, not storing data unencrypted)

Used 1 reference

```
# Python script to collect user data: nam
# Note: To protect user privacy, do not s
# You can anonymize emails by hashing, an

import hashlib

name = input("Enter your name: ")
age = int(input("Enter your age: "))
email = input("Enter your email: ")

# Hash the email to anonymize it before s
hashed_email = hashlib.sha256(email.encod

print("\n--- User Data Collected ---")
print(f"Name: {name}") # Consider using
print(f"Age: {age}")
print(f"Email (hashed): {hashed_email}")

# Tip: Never store raw emails or other sa
```

TASK-2:

Prompt :

generate a Python function for sentiment analysis.

Code:

```
from nltk.sentiment import SentimentIntensityAnalyzer
import nltk

# Download VADER lexicon (only needed once)

nltk.download('vader_lexicon')

def analyze_sentiment_vader(text):
    """
    Analyzes sentiment using VADER from NLTK.

    Args:
        text (str): Input text to analyze.

    Returns:
        dict: Contains compound score and sentiment label.
    """
    sia = SentimentIntensityAnalyzer()
    scores = sia.polarity_scores(text)
    compound = scores['compound']

    # Interpret compound score

    if compound ≥ 0.05:
        sentiment = "Positive"
    elif compound ≤ -0.05:
        sentiment = "Negative"
    else:
        sentiment = "Neutral"

    return {
        "compound_score": compound,
        "sentiment": sentiment,
```

```

        "sentiment": sentiment,
        "detailed_scores": scores
    }

# Example usage

text = "The visuals were stunning, but the gameplay felt repetitive."
result = analyze_sentiment_vader(text)
print(result)

```

Output Example

Python

 Copy

```

{
  'compound_score': 0.4215,
  'sentiment': 'Positive',
  'detailed_scores': {
    'neg': 0.204,
    'neu': 0.584,
    'pos': 0.212,
    'compound': 0.4215
  }
}

```

Prompt:

identify and handle potential biases in the data at which the output should be like Copilot-generated code with additions or comments addressing bias mitigation strategies (e.g., balancing dataset, removing offensive terms)

code:

Python

 Copy

```

import pandas as pd
import numpy as np
import re
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.utils import resample
from sklearn.metrics import classification_report

```

TASK-3:

Prompt:

write a Python program that recommends products based on user history.and also include explanations, fairness checks (e.g., avoiding favoritism), and user feedback options in the code

code:

Python

 Copy

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

# Step 1: Sample product catalog

products = pd.DataFrame({
    'product_id': [101, 102, 103, 104, 105],
    'name': ['Eco Bottle', 'Smart Watch', 'Yoga Mat', 'Gaming Mouse', 'Organic Tea'],
    'category': ['Home', 'Electronics', 'Fitness', 'Electronics', 'Food'],
    'description': [
        'Reusable eco-friendly water bottle',
        'Fitness tracker with heart rate monitor',
        'Non-slip mat for yoga and pilates',
        'High precision mouse for gaming',
        'Herbal tea with natural ingredients'
    ],
    'popularity_score': [0.8, 0.95, 0.6, 0.9, 0.7] # Used for fairness check
})

# Step 2: User history (simulate past purchases or likes)
user_history = ['Smart Watch', 'Gaming Mouse']

# Step 3: Vectorize product descriptions
vectorizer = TfidfVectorizer()
tfidf_matrix = vectorizer.fit_transform(products['description'])

# Step 4: Compute similarity between user history and all products
```

```

def get_user_profile(history, product_df, tfidf_matrix):
    history_indices = product_df[product_df['name'].isin(history)].index
    user_vector = tfidf_matrix[history_indices].mean(axis=0)
    return user_vector

user_profile = get_user_profile(user_history, products, tfidf_matrix)
similarities = cosine_similarity(user_profile, tfidf_matrix).flatten()

# Step 5: Fairness check – avoid over-recommending popular items

def apply_fairness(similarities, product_df):
    fairness_weights = 1 - product_df['popularity_score'] # Penalize high-po

    adjusted_scores = similarities * fairness_weights
    return adjusted_scores

adjusted_scores = apply_fairness(similarities, products)

# Step 6: Generate recommendations

products['score'] = adjusted_scores
recommended = products.sort_values(by='score', ascending=False).head(3)

# Step 7: Display recommendations with explanations

print("🔍 Recommended Products Based on Your Interests:\n")
for _, row in recommended.iterrows():
    print(f"– {row['name']} ({row['category']})")
    print(f"  Why: Similar to your interest in {'', '.join(user_history)}")
    print(f"  Description: {row['description']}")
    print(f"  Fairness-adjusted score: {round(row['score'], 3)}\n")

# Step 8: Collect user feedback

def collect_feedback():
    print("💬 Did you like these recommendations? (yes/no)")
    feedback = input().strip().lower()
    if feedback == 'no':
        print("🔧 We'll adjust future suggestions. You can also specify what")
    else:
        print("✅ Great! We'll keep improving your experience.")

collect_feedback()

```

TASK-4:

Prompt:

generate logging functionality in a Python web application.

Code:

```

import logging
from flask import Flask, request, jsonify

app = Flask(__name__)

# Step 1: Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s [%(levelname)s] %(message)s',
    handlers=[
        logging.FileHandler("app.log"),          # Logs to file
        logging.StreamHandler()                  # Logs to console
    ]
)

logger = logging.getLogger(__name__)

# Step 2: Middleware to log each request
@app.before_request
def log_request_info():
    logger.info(f"Incoming request: {request.method} {request.path}")
    # Avoid logging sensitive data like passwords or tokens

    if request.method == 'POST':
        logger.info(f"Request data: {sanitize(request.json)}")

def sanitize(data):
    """Remove sensitive fields before logging."""

```

```

        if not data:
            return {}
        sensitive_keys = ['password', 'token']
        return {k: '[REDACTED]' if k in sensitive_keys else v for k, v in data.items()}

# Step 3: Sample route with logging and feedback
@app.route('/recommend', methods=['POST'])
def recommend():
    user_input = request.json.get('history', [])
    logger.info(f"Generating recommendations for: {user_input}")

    # Simulate recommendation logic

    recommendations = ["Eco Bottle", "Yoga Mat", "Organic Tea"]
    logger.info(f"Recommended: {recommendations}")

    return jsonify({
        "recommendations": recommendations,
        "message": "Recommendations generated successfully."
    })

# Step 4: Feedback route
@app.route('/feedback', methods=['POST'])
def feedback():
    user_feedback = request.json.get('feedback', '')
    logger.info(f"User feedback received: {user_feedback}")
    return jsonify({"status": "Thank you for your feedback!"})

# Step 5: Error handling
@app.errorhandler(Exception)
def handle_exception(e):

```

```

    logger.error(f"Unhandled exception: {str(e)}", exc_info=True)
    return jsonify({"error": "Something went wrong."}), 500

if __name__ == '__main__':
    app.run(debug=True)

```

Prompt:

ensure the logs do not record sensitive information like Logging code that avoids saving personal identifiers (e.g., passwords, emails), and includes comments about ethical logging practices.

Code:


```

import logging
from flask import Flask, request, jsonify

app = Flask(__name__)

# Step 1: Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s [%(levelname)s] %(message)s',
    handlers=[
        logging.FileHandler("app.log"),      # Logs to file
        logging.StreamHandler()              # Logs to console
    ]
)

logger = logging.getLogger(__name__)

# Step 2: Define sanitization function
def sanitize(data):
    """
    Removes or redacts sensitive fields from incoming data before logging.
    Ethical logging practice: Never store personal identifiers like passwords
    """
    if not data:
        return {}

    sensitive_keys = ['password', 'email', 'token', 'ssn', 'credit_card']
    sanitized = {}

```

```

    for key, value in data.items():
        if key.lower() in sensitive_keys:
            sanitized[key] = '[REDACTED]'
        else:
            sanitized[key] = value
    return sanitized

# Step 3: Log incoming requests safely

@app.before_request
def log_request():
    logger.info(f"Request: {request.method} {request.path}")

    # Log sanitized body only for POST/PUT

    if request.method in ['POST', 'PUT']:
        try:
            raw_data = request.get_json(force=True)
            safe_data = sanitize(raw_data)
            logger.info(f"Sanitized Data: {safe_data}")
        except Exception as e:
            logger.warning(f"Failed to parse request body: {str(e)}")

# Step 4: Sample route

@app.route('/register', methods=['POST'])
def register():
    data = request.get_json()
    # Business logic here ...

    return jsonify({"status": "User registered successfully."})

# Step 5: Error handling

```

```

@app.errorhandler(Exception)
def handle_error(e):
    logger.error(f"Unhandled exception: {str(e)}", exc_info=True)
    return jsonify({"error": "Internal server error"}), 500

if __name__ == '__main__':
    app.run(debug=True)

```

Task-5:

Prompt:

generate a machine learning model.

Code:

```

import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report

# Step 1: Sample dataset
data = pd.DataFrame({
    'price': [199, 99, 299, 149, 89],
    'rating': [4.5, 3.8, 4.9, 4.2, 3.5],
    'category': ['electronics', 'home', 'electronics', 'fitness', 'home'],
    'liked': [1, 0, 1, 1, 0] # Target variable
})

# Step 2: Encode categorical features
data = pd.get_dummies(data, columns=['category'], drop_first=True)

# Step 3: Split and scale
X = data.drop('liked', axis=1)
y = data['liked']

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2)

# Step 4: Train model
model = LogisticRegression()
model.fit(X_train, y_train)

```

```

model.fit(X_train, y_train)

# Step 5: Evaluate
y_pred = model.predict(X_test)
print(classification_report(y_test, y_pred))

```

Prompt:

add documentation on how to use the model responsibly (e.g., explainability, accuracy limits) like code with a README or inline documentation suggesting responsible usage, limitations, and fairness considerations

Code:

```

import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report

# Step 1: Create synthetic dataset

data = pd.DataFrame({
    'price': [199, 99, 299, 149, 89],
    'rating': [4.5, 3.8, 4.9, 4.2, 3.5],
    'category': ['electronics', 'home', 'electronics', 'fitness', 'home'],
    'liked': [1, 0, 1, 1, 0] # Target variable
})

# Step 2: Encode categorical features

# Ethical note: Avoid encoding sensitive attributes unless necessary and just

data = pd.get_dummies(data, columns=['category'], drop_first=True)

# Step 3: Split and scale features

X = data.drop('liked', axis=1)
y = data['liked']

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2)

# Step 4: Train interpretable model

```

```

model = LogisticRegression()
model.fit(X_train, y_train)

# Step 5: Evaluate model performance

y_pred = model.predict(X_test)
print("📊 Model Evaluation Report:")
print(classification_report(y_test, y_pred))

# Step 6: Inspect model coefficients for transparency

print("🔍 Feature Influence (Coefficients):")
for feature, coef in zip(X.columns, model.coef_[0]):
    print(f"{feature}: {round(coef, 3)}")

```