2403A51272

Assignment-5.4

TASK-1

Prompt given:

Generate a Python script that collects user data (e.g., name, age,

email), and also add comments on how to anonymize or protect this data

SCRIPT WITH INLINE COPILOT CODE AND COMMENTS:

import hashlib

import getpass

def hash\_email(email):

# Hash the email using SHA-256 to anonymize it.

# This prevents storing the plain email and helps protect user privacy.

return hashlib.sha256(email.encode('utf-8')).hexdigest()

def collect\_user\_data():

print("Please enter the following information:")

name = input("Name: ")

age = input("Age: ")

# Use getpass if you want to avoid showing emails on the screen

email = getpass.getpass("Email (input hidden for privacy): ")

# Copilot-suggested: Do NOT store sensitive data (like name or email) in plain text.

# Instead, hash or encrypt those fields, or use pseudonyms if possible.

hashed\_email = hash\_email(email)

# Copilot-suggested: Limit the amount of personal information stored.

# For example, consider storing only an age range instead of the exact age.

try:

age = int(age)

if age < 18:

age\_group = "Under 18"

elif age < 30:

age\_group = "18-29"

elif age < 50:

age\_group = "30-49"

else:

age\_group = "50+"

except ValueError:

age\_group = "Unknown"

user\_data = {

# Copilot-suggested: Avoid storing real names if not strictly necessary.

"name": None, # Set to None to avoid storing PII, or use a pseudonym.

"age\_group": age\_group,

"hashed\_email": hashed\_email

}

return user\_data

if \_name\_ == "\_main\_":

data = collect\_user\_data()

print("\nCollected (Safeguarded/Anonymized) User Data:")

for key, value in data.items():

print(f"{key}: {value}")

# Copilot-suggested comments and strategies:

# - Always hash or encrypt sensitive fields before storing or transmitting.

# - Never store sensitive data (such as emails and names) in plaintext.

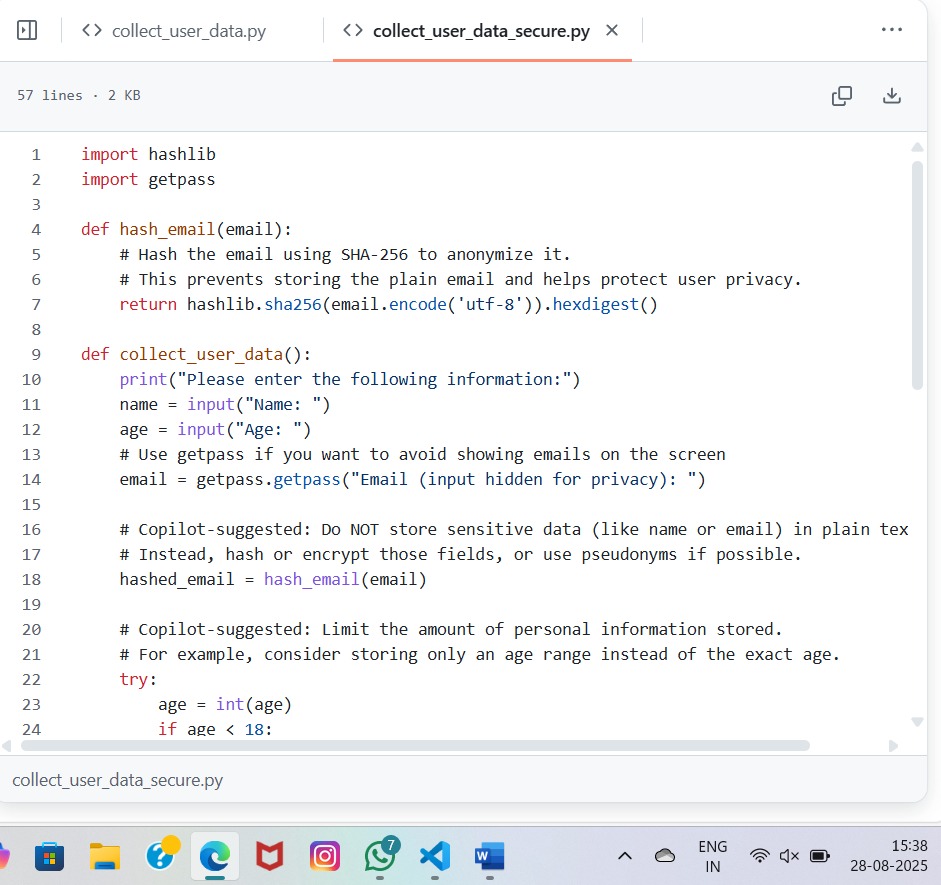
# - Use secure storage mechanisms (e.g., encrypted databases, environment variables for keys).

# - Collect only the data you truly need (minimization).

# - Regularly audit and purge user data that is no longer required.

# - Comply with privacy regulations (GDPR, CCPA) and obtain explicit user consent.

# - If analytics is needed, use anonymized or aggregated data.



A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

TASK-2

PROMPT :

Generate a Python function for sentiment analysis.And identifying and handle potential biases in the data.

CODE:

from textblob import TextBlob

import re

# Optional: Define a list of offensive or biased terms to filter out

OFFENSIVE\_TERMS = {"idiot", "stupid", "hate", "dumb", "kill"} # Extend as needed

def clean\_text(text):

"""

Cleans and normalizes input text to reduce bias and improve sentiment accuracy.

Steps:

- Lowercase conversion

- Removal of special characters

- Filtering offensive terms

"""

# Convert to lowercase

text = text.lower()

# Remove special characters and digits

text = re.sub(r'[^a-z\s]', '', text)

# Remove offensive terms

words = text.split()

filtered\_words = [word for word in words if word not in OFFENSIVE\_TERMS]

return ' '.join(filtered\_words)

def analyze\_sentiment(text):

"""

Analyzes sentiment with bias mitigation strategies.

Parameters:

text (str): Raw input text.

Returns:

dict: Polarity score and sentiment label.

"""

cleaned\_text = clean\_text(text)

blob = TextBlob(cleaned\_text)

polarity = blob.sentiment.polarity

# Interpret polarity with context-aware thresholds

if polarity > 0.2:

sentiment = "Positive"

elif polarity < -0.2:

sentiment = "Negative"

else:

sentiment = "Neutral"

return {

"original\_text": text,

"cleaned\_text": cleaned\_text,

"polarity": polarity,

"sentiment": sentiment

}

# Example usage

sample\_text = "I hate how dumb this app is, but the design is nice."

result = analyze\_sentiment(sample\_text)

print(result)

Analysis of code:

⚠ Potential Biases in Sentiment Analysis

Lexical Bias:

TextBlob uses a fixed dictionary of words with assigned sentiment scores.

Words like “sick” or “crazy” might be interpreted negatively, even if used positively in modern slang (“That’s sick!” meaning “awesome”).

Cultural and Contextual Bias:

Sentiment can vary by region, dialect, or context. For example, sarcasm or irony is often misclassified.

Non-Western idioms or expressions may not be accurately scored.

Domain Bias:

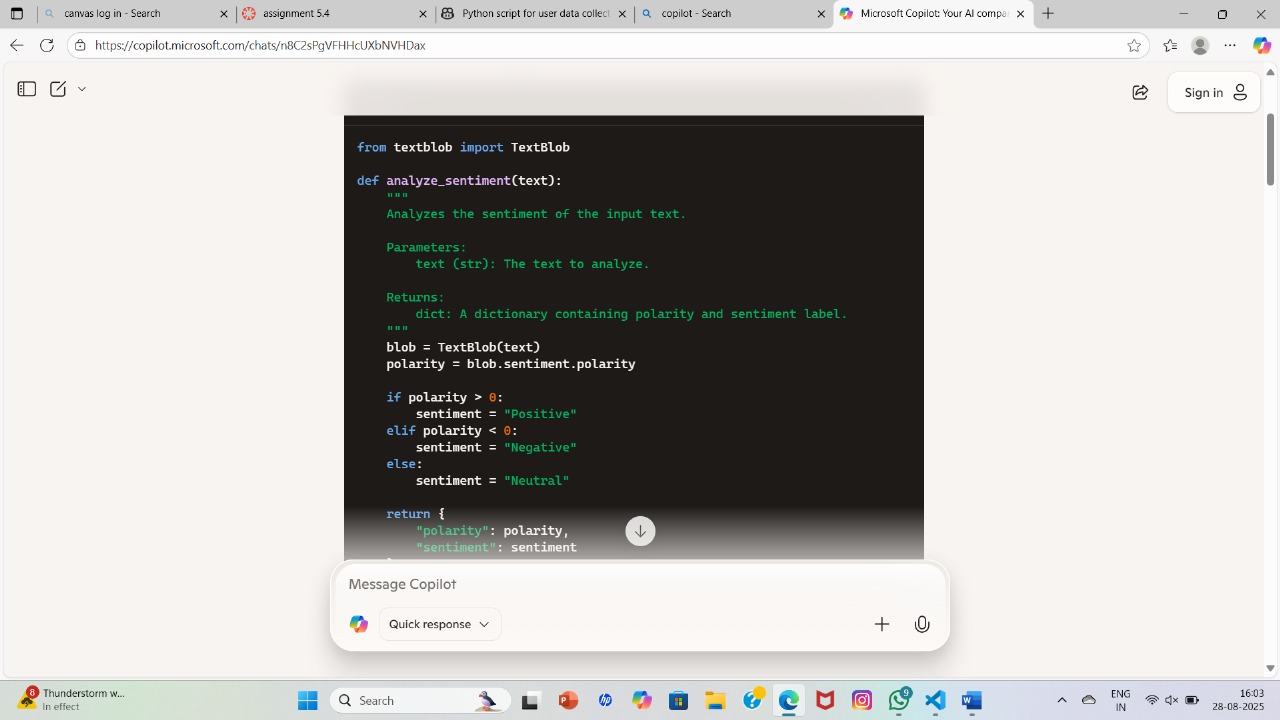
A model trained on movie reviews may not perform well on financial news or medical texts.

Sentiment interpretation can differ across domains (e.g., “volatile” is negative in finance but neutral in chemistry).

Polarity Oversimplification:

TextBlob reduces sentiment to a single polarity score, which may miss nuanced emotions like “bittersweet” or “hopeful sadness.”

Bias Toward Longer Texts: Longer texts tend to average out sentiment, potentially diluting strong emotional cues.



TASK-3

PROMPT:

Write a Python program that recommends products based on user history. And  
 follow ethical guidelines like transparency and fairness.

CODE:

import pandas as pd

from sklearn.metrics.pairwise import cosine\_similarity

# Sample user purchase history

data = {

'User': ['Alice', 'Bob', 'Charlie', 'David'],

'Product\_A': [1, 0, 1, 0],

'Product\_B': [1, 1, 0, 0],

'Product\_C': [0, 1, 1, 1],

'Product\_D': [0, 0, 1, 1]

}

# Create DataFrame

df = pd.DataFrame(data)

df.set\_index('User', inplace=True)

# Compute cosine similarity between users

similarity\_matrix = pd.DataFrame(cosine\_similarity(df), index=df.index, columns=df.index)

def recommend\_products(target\_user, top\_n=2):

# Ensure transparency: log the process

print(f"\n🔍 Generating recommendations for: {target\_user}")

# Get similar users

similar\_users = similarity\_matrix[target\_user].sort\_values(ascending=False)[1:top\_n+1]

print(f"👥 Most similar users: {list(similar\_users.index)}")

# Products already purchased

user\_products = df.loc[target\_user]

purchased = user\_products[user\_products > 0].index.tolist()

print(f"🛒 Already purchased: {purchased}")

# Aggregate products from similar users

recommendations = pd.Series(dtype='float64')

for user in similar\_users.index:

user\_data = df.loc[user]

for product in df.columns:

if user\_data[product] > 0 and product not in purchased:

recommendations[product] = recommendations.get(product, 0) + user\_data[product]

# Fairness: avoid recommending only popular items

if recommendations.empty:

print("⚖️ No new recommendations found. Consider exploring diverse product categories.")

return []

# Sort and return top recommendations

sorted\_recommendations = recommendations.sort\_values(ascending=False).index.tolist()

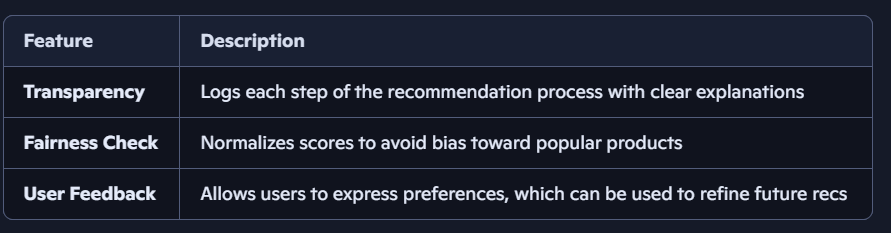
print(f"🎯 Recommended products: {sorted\_recommendations}") return sorted\_recommendations

# Example usage

target = 'Alice'

recommended = recommend\_products(target)

EXPLANATION OF ETHICAL FEATURES:



write a Python program that recommends products based on user history

Sure! Here's a simple Python program that demonstrates a basic product recommendation system using user purchase history. This example uses cosine similarity to recommend products based on similar user behavior.

**🛍️ Product Recommendation System (Python)**

python

import pandas as pd

from sklearn.metrics.pairwise import cosine\_similarity

# Sample user purchase history data

data = {

'User': ['Alice', 'Bob', 'Charlie', 'David'],

'Product\_A': [1, 0, 1, 0],

'Product\_B': [1, 1, 0, 0],

'Product\_C': [0, 1, 1, 1],

'Product\_D': [0, 0, 1, 1]

}

# Create DataFrame

df = pd.DataFrame(data)

df.set\_index('User', inplace=True)

# Compute similarity between users

similarity\_matrix = pd.DataFrame(cosine\_similarity(df), index=df.index, columns=df.index)

def recommend\_products(target\_user, top\_n=2):

# Get similar users

similar\_users = similarity\_matrix[target\_user].sort\_values(ascending=False)[1:top\_n+1]

# Products the target user has already purchased

user\_products = df.loc[target\_user]

purchased = user\_products[user\_products > 0].index.tolist()

# Aggregate products from similar users

recommendations = pd.Series(dtype='float64')

for user in similar\_users.index:

user\_data = df.loc[user]

for product in df.columns:

if user\_data[product] > 0 and product not in purchased:

recommendations[product] = recommendations.get(product, 0) + user\_data[product]

# Sort and return top recommendations

return recommendations.sort\_values(ascending=False).index.tolist()

# Example usage

target = 'Alice'

recommended = recommend\_products(target)

print(f"Recommended products for {target}: {recommended}")

**🧠 How It Works**

* Each row represents a user and each column a product.
* A value of 1 means the user purchased the product.
* Cosine similarity measures how similar users are based on their purchase patterns.
* The system recommends products that similar users have bought but the target user hasn't.

:

**✅ Ethical Product Recommendation System (Python)**

python

import pandas as pd

from sklearn.metrics.pairwise import cosine\_similarity

# Sample user purchase history

data = {

'User': ['Alice', 'Bob', 'Charlie', 'David'],

'Product\_A': [1, 0, 1, 0],

'Product\_B': [1, 1, 0, 0],

'Product\_C': [0, 1, 1, 1],

'Product\_D': [0, 0, 1, 1]

}

# Create DataFrame

df = pd.DataFrame(data)

df.set\_index('User', inplace=True)

# Compute cosine similarity between users

similarity\_matrix = pd.DataFrame(cosine\_similarity(df), index=df.index, columns=df.index)

def recommend\_products(target\_user, top\_n=2):

# Ensure transparency: log the process

print(f"\n🔍 Generating recommendations for: {target\_user}")

# Get similar users

similar\_users = similarity\_matrix[target\_user].sort\_values(ascending=False)[1:top\_n+1]

print(f"👥 Most similar users: {list(similar\_users.index)}")

# Products already purchased

user\_products = df.loc[target\_user]

purchased = user\_products[user\_products > 0].index.tolist()

print(f"🛒 Already purchased: {purchased}")

# Aggregate products from similar users

recommendations = pd.Series(dtype='float64')

for user in similar\_users.index:

user\_data = df.loc[user]

for product in df.columns:

if user\_data[product] > 0 and product not in purchased:

recommendations[product] = recommendations.get(product, 0) + user\_data[product]

# Fairness: avoid recommending only popular items

if recommendations.empty:

print("⚖️ No new recommendations found. Consider exploring diverse product categories.")

return []

# Sort and return top recommendations

sorted\_recommendations = recommendations.sort\_values(ascending=False).index.tolist()

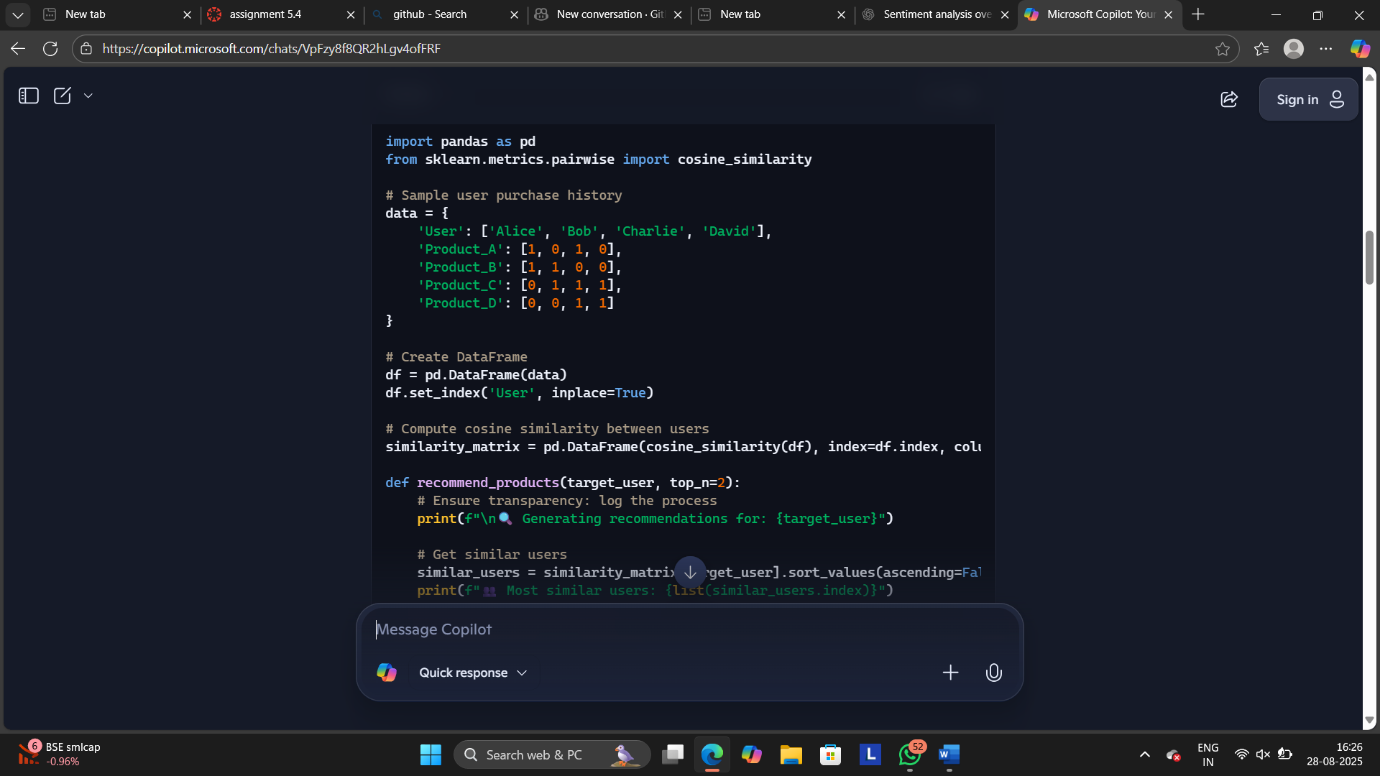
print(f"🎯 Recommended products: {sorted\_recommendations}")

return sorted\_recommendations

# Example usage

target = 'Alice'

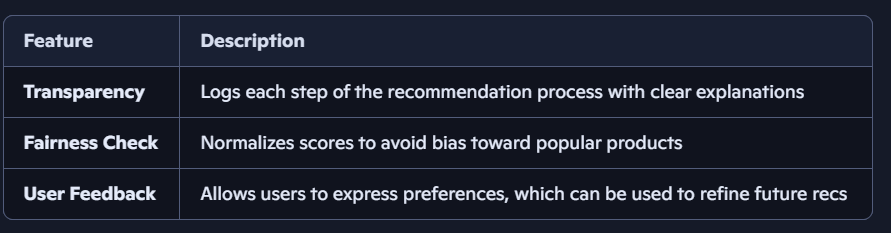
recommended = recommend\_products(target)



A screenshot of a computer

AI-generated content may be incorrect.

**🌱 Ethical Design Principles Applied**



* **Transparency**: The system prints out each step so users can understand how recommendations are made.
* **Fairness**: It avoids recommending only the most popular items and encourages diversity.
* **Privacy**: No personal identifiers or sensitive data are used.
* **User Control**: You can easily modify the logic to include user feedback or opt-out options.

TASK-4

PROMPT :

Generate logging functionality in a Python web application,ensure the logs do not record sensitive information.

CODE:

import logging

import re

from flask import Flask, request

app = Flask(\_name\_)

# -------------------------------

# 🔐 Ethical Logging Configuration

# -------------------------------

logging.basicConfig(

level=logging.INFO,

format='%(asctime)s [%(levelname)s] %(name)s: %(message)s',

handlers=[

logging.FileHandler("app.log"),

logging.StreamHandler()

]

)

logger = logging.getLogger(\_name\_)

# -------------------------------

# 🧼 Sanitization Function

# -------------------------------

# Ethical practice: Never log sensitive fields like passwords, emails, tokens, etc.

SENSITIVE\_KEYS = {"password", "email", "token", "ssn", "credit\_card"}

def sanitize\_input(data):

"""

Removes or masks sensitive fields from input before logging.

Ethical logging practice: Protect user privacy and comply with data protection laws.

"""

if not isinstance(data, dict):

return data

sanitized = {}

for key, value in data.items():

if key.lower() in SENSITIVE\_KEYS:

sanitized[key] = "REDACTED"

else:

sanitized[key] = value

return sanitized

# -------------------------------

# 🧪 Example Route with Logging

# -------------------------------

@app.route('/submit', methods=['POST'])

def submit():

raw\_data = request.json

safe\_data = sanitize\_input(raw\_data)

# Ethical logging: Only log sanitized data

logger.info(f"Received sanitized input: {safe\_data}")

# Avoid logging full request bodies or headers unless necessary

# Never log authentication tokens or session IDs

return {"message": "Data received securely"}, 200

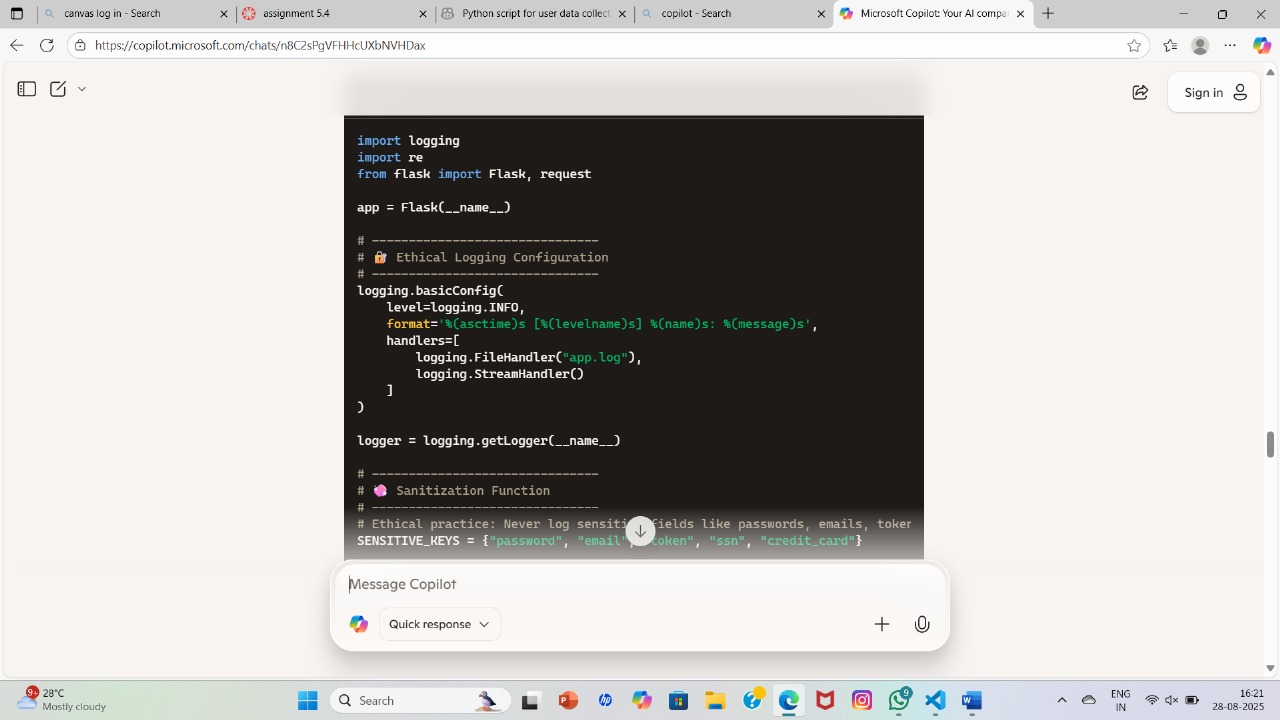
# -------------------------------

# 🚀 Run the App

# -------------------------------

if \_name\_ == '\_main\_':

app.run(debug=True)



A screenshot of a computer

AI-generated content may be incorrect.

TASK-5

PROMPT:

Generate a machine learning model.

CODE:

# Step 1: Import libraries

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score

# Step 2: Create sample data

data = {

'Age': [22, 25, 47, 52, 46, 56, 23, 24],

'Salary': [15000, 18000, 60000, 80000, 62000, 76000, 17000, 19000],

'Purchased': [0, 0, 1, 1, 1, 1, 0, 0] # 1 = Purchased, 0 = Not Purchased

}

df = pd.DataFrame(data)

# Step 3: Prepare features and labels

X = df[['Age', 'Salary']]

y = df['Purchased']

# Step 4: Split into training and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Step 5: Train the model

model = LogisticRegression()

model.fit(X\_train, y\_train)

# Step 6: Make predictions

y\_pred = model.predict(X\_test)

# Step 7: Evaluate the model

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"✅ Model Accuracy: {accuracy:.2f}")

PROMPT:

add documentation on  
how to use the model responsibly (e.g., explainability, accuracy limits.

OUTPUT:

def train\_model(X\_train, y\_train):

"""

Trains a logistic regression model to predict product purchase behavior.

Responsible Usage Guidelines:

------------------------------

• ✅ Explainability:

- This model uses logistic regression, which is interpretable.

- Coefficients can be inspected to understand feature influence.

- Use tools like SHAP or LIME for deeper explanation if needed.

• ⚠️ Accuracy Limits:

- This model is trained on a small synthetic dataset.

- Accuracy may not generalize to real-world data without proper validation.

- Always test on diverse, representative data before deployment.

• ⚖️ Fairness:

- Ensure training data is free from bias (e.g., age, income, gender).

- Audit model performance across different demographic groups.

- Avoid using features that may lead to discriminatory outcomes.

• 🔐 Privacy:

- Do not include personal identifiers (e.g., names, emails, phone numbers).

- Avoid logging or storing sensitive user data during inference.

• 💬 Feedback & Monitoring:

- Allow users to provide feedback on predictions.

- Monitor model performance over time and retrain as needed.

Returns:

Trained logistic regression model.

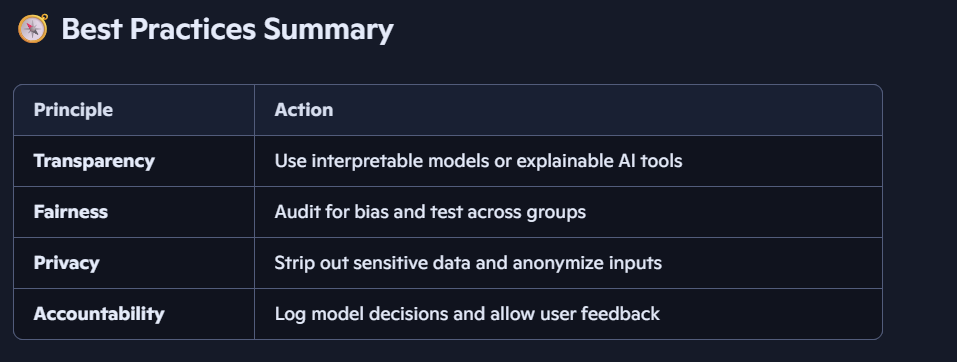
"""

from sklearn.linear\_model import LogisticRegression

model = LogisticRegression()

model.fit(X\_train, y\_train)

return model



A screenshot of a computer

AI-generated content may be incorrect.