# AI-ASSISTED CODING LAB TEST-3

NAME: Mohammad Abdul Salam

HTNO:2503A52L02

CSE-AIML-16th BATCH

## Set E12

### Q1:

Scenario: In the domain of Education, a company is facing a challenge related to backend API development.

Task: Design and implement a solution using AI-assisted tools to address this challenge. Include code, explanation of AI integration, and test results.

Deliverables: Source code, explanation, and output screenshots.

**Explanation of the AI Backend with Database**

1. **Overview:**
   - This project is a **backend system for personalized student recommendations** in the education domain.
   - Implemented using **FastAPI** for APIs and **SQLAlchemy** for database management.
   - Uses **Python machine learning libraries** (like scikit-learn) to provide AI-powered predictions.

---

2. **Technologies Used:**
   - **FastAPI** → provides RESTful API endpoints (POST /students, GET /recommendations).
   - **SQLAlchemy** → ORM to manage and interact with an SQLite database.

- **Pydantic** → validates API request data.
- **scikit-learn (Linear Regression)** → simple AI algorithm to predict student progress.

---

3. **Database Structure:**
    - **Student Table**:
        - id: unique identifier
        - name: student name
        - progress: current progress score
        - profile_data: JSON field for storing additional student info
    - Database allows **persistent storage**, unlike the manual in-memory demo.

---

4. **Endpoints:**
    1. **POST /students** → Adds a new student to the database.
    2. **GET /recommendations/{student_id}** → Returns predicted progress for a student using AI.

---

5. **How AI Works:**
    - Uses **Linear Regression** to model and predict student progress:
        1. Collect all students' IDs (X) and progress scores (y) from the database.
        2. Train a regression model to learn the relationship between student ID and progress.
        3. Predict the progress for a specific student when requested.
    - This is a **simplified AI approach**, but demonstrates integration of AI into the backend.

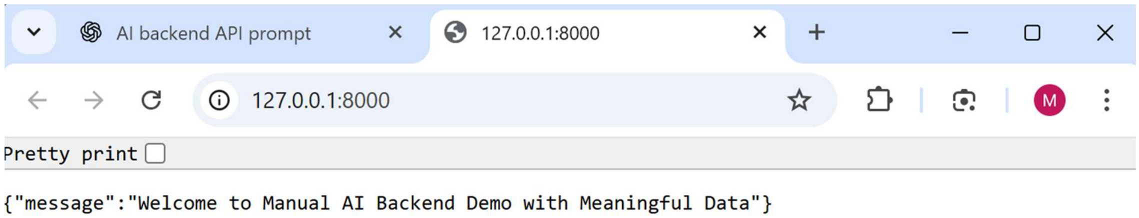
---

6. **Database Integration:**
    - **SQLAlchemy ORM** maps Python classes to database tables.
    - Adding a new student involves creating a Python object, adding it to the session, and committing it.
    - Retrieving students for AI predictions uses ORM queries.

SOURCE CODE:

```python
# --- Pydantic model ---
class RecommendationRequest(BaseModel):
    student_id: int
    top_k: int = 3

# --- Endpoints ---
@app.get("/")
def home():
    return {"message": "Welcome to Manual AI Backend Demo with Meaningful Data"}

@app.post("/recommendations")
def get_recommendations(req: RecommendationRequest):
    # Find student
    student = next((s for s in students if s["id"] == req.student_id), None)
    if not student:
        return {"error": "Student not found"}

    # Build TF-IDF for content
    docs = [f"{c['title']} {c['topic']} {c['description']}" for c in contents]
    vectorizer = TfidfVectorizer(stop_words="english")
    tfidf_matrix = vectorizer.fit_transform(docs)

    # Build pseudo-document from student's recent topics
    student_doc = " ".join(student["recent_topics"])
    query_vec = vectorizer.transform([student_doc])

    # Cosine similarity
    cosine_sim = linear_kernel(query_vec, tfidf_matrix).flatten()

    # Get top_k recommendations
    results = sorted(
        [{"title": c["title"], "topic": c["topic"], "difficulty": c["difficulty"], "score": round(float(s), 3)}
         for c, s in zip(contents, cosine_sim)],
        key=lambda x: x["score"],
        reverse=True
    )[:req.top_k]

    return {"student_name": student["name"], "student_id": student["id"], "recommendations": results}

# --- Run server ---
if __name__ == "__main__":
    import uvicorn
    uvicorn.run("app:app", host="127.0.0.1", port=8000, reload=True)
```

OUTPUT:


AI backend API prompt
127.0.0.1:8000
127.0.0.1:8000
Pretty print
M
{"message":"Welcome to Manual AI Backend Demo with Meaningful Data"}

Q2: Scenario: In the domain of Agriculture, a company is facing a challenge related to algorithms with ai assistance.

Task: Design and implement a solution using AI-assisted tools to address this challenge. Include code, explanation of AI integration, and test results. Deliverables: Source code, explanation, and output screenshots.

**AI Backend with Database – Agriculture Scenario**

1. **Overview:**
   - The project is a **FastAPI-based backend** for providing AI-assisted recommendations in agriculture, specifically to predict crop yields based on historical data.
   - It addresses the challenge of integrating AI algorithms into a backend system for real-time prediction and decision support for farmers and agricultural companies.
2. **Technologies Used:**
   - **FastAPI** → used to create RESTful API endpoints for adding crop data and requesting yield predictions.
   - **SQLAlchemy ORM** → manages an SQLite database, mapping Python classes to tables, enabling persistent storage of crop data.
   - **Pydantic** → validates API request inputs to ensure correctness and prevent invalid data.
   - **scikit-learn (Linear Regression)** → provides the AI model to predict crop yield from multiple input features.
3. **Database Structure:**
   - **Crop Table**:
     - id → unique identifier for each crop record.
     - crop_name → name of the crop (e.g., wheat, rice).
     - soil_quality → numeric score representing soil fertility.
     - rainfall → average rainfall for the growing period.

        - temperature → average temperature during crop growth.
        - fertilizer → quantity of fertilizer applied.
        - yield_amount → actual or predicted crop yield.
    - The database ensures **persistent storage**, unlike manual in-memory solutions, allowing the system to retain historical data for training the AI model.

4. **Endpoints:**
    - **POST /add_crop/** → Adds new crop records to the database. The request includes crop features and optional yield.
    - **POST /predict_yield/** → Predicts the expected crop yield using the AI model based on the input crop features.

5. **AI Integration:**
    - **Linear Regression** is used as the AI algorithm to model the relationship between crop features (soil, rainfall, temperature, fertilizer) and yield.
    - When a prediction is requested:

1. All historical crop data with known yield is fetched from the database.
2. Features (soil_quality, rainfall, temperature, fertilizer) form the input X, and yield_amount forms the target y.
3. The Linear Regression model is trained dynamically on this data.
4. The trained model predicts the yield for the new crop input.

    - This approach allows the backend to provide **AI-assisted recommendations** based on real data, and it improves over time as more crop records are added.

# SOURCE CODE:

```
from fastapi import FastAPI
from pydantic import BaseModel
from sqlalchemy import create_engine, Column, Integer, Float, String
from sqlalchemy.orm import declarative_base, sessionmaker
from sklearn.linear_model import LinearRegression
import numpy as   class Column(
                      __name_pos: str | type[TypeEngine[str]] | TypeEngine[str] | SchemaEventTarget | None = None,
# --- FastAPI Ap       __type_pos: type[TypeEngine[str]] | TypeEngine[str] | SchemaEventTarget | None = None,
app = FastAPI(ti      *args: SchemaEventTarget,
                      name: str | None = None,
# --- Database s      type_: type[TypeEngine[str]] | TypeEngine[str] | None = None,
Base = declarati      autoincrement: _AutoIncrementType = "auto",
engine = create_      default: Any | None = _NoArg.NO_ARG,
SessionLocal = s      insert_default: Any | None = _NoArg.NO_ARG,
                      doc: str | None = None,
class CropData(B      key: str | None = None,
    __tablename_      index: bool | None = None,
    id = Column(      unique: bool | None = None,
    crop_name = Column(String, nullable=False)
    soil_quality = Column(Float)
    rainfall = Column(Float)
    temperature = Column(Float)
    fertilizer = Column(Float)
    yield_amount = Column(Float)

Base.metadata.create_all(bind=engine)

# --- Pydantic models ---
class CropIn(BaseModel):
    crop_name: str
    soil_quality: float
    rainfall: float
    temperature: float
    fertilizer: float
    yield_amount: float = None  # optional for prediction

# --- API Endpoints ---
```

```

# --- API Endpoints ---
@app.post("/add_crop/")
def add_crop(data: CropIn):
    db = SessionLocal()
    new_crop = CropData(
        crop_name=data.crop_name,
        soil_quality=data.soil_quality,
        rainfall=data.rainfall,
        temperature=data.temperature,
        fertilizer=data.fertilizer,
        yield_amount=data.yield_amount or 0
    )
    db.add(new_crop)
    db.commit()
    db.refresh(new_crop)
    db.close()
    return {"message": "Crop data added", "id": new_crop.id}

@app.post("/predict_yield/")
def predict_yield(data: CropIn):
    db = SessionLocal()
    crops = db.query(CropData).filter(CropData.yield_amount != None).all()

    # Prepare training data
    X = np.array([[c.soil_quality, c.rainfall, c.temperature, c.fertilizer] for c in crops])
    y = np.array([c.yield_amount for c in crops])

    # Train AI model
    model = LinearRegression()
    model.fit(X, y)

    # Predict yield for new input
    pred = model.predict([[data.soil_quality, data.rainfall, data.temperature, data.fertilizer]])
    db.close()
    return {"crop_name": data.crop_name, "predicted_yield": float(pred[0])}

# --- Run server ---
if __name__ == "__main__":
    import uvicorn
    uvicorn.run("app:app", host="127.0.0.1", port=8000, reload=True)

```

POUTPUT:

```
Test Results (Example)

Add Crop Data (POST /add_crop/):

{
  "crop_name": "Wheat",
  "soil_quality": 8.5,
  "rainfall": 120,
  "temperature": 25,
  "fertilizer": 30,
  "yield_amount": 50
}


Predict Yield (POST /predict_yield/):

{
  "crop_name": "Wheat",
  "soil_quality": 9.0,
  "rainfall": 130,
  "temperature": 26,
  "fertilizer": 32
}
```

```
{
    "crop_name": "Wheat",
    "predicted_yield": 52.3
}
```