Part 1: Short Answer Questions

1. Problem Definition

Hypothetical AI Problem: Predicting student dropout rates in universities.

**Objectives:*

Identify at-risk students early in the academic year using historical and real-time data.

Recommend personalized interventions to reduce dropout likelihood.

Improve overall student retention rates over the next academic year.

**Stakeholders:*

University administrators (e.g., Deans, Academic Advisors)

Students and their families

Key Performance Indicator (KPI):

Dropout rate reduction percentage — measures the change in dropout rates before and after AI-based interventions.

2. Data Collection & Preprocessing

Data Sources:

*University Student Information System (SIS)

Includes academic records (e.g., GPA, attendance, course registration, credit load), demographic details (e.g., age, gender), and enrollment history.

*Learning Management System (LMS) Logs

Contains data on student engagement, such as login frequency, assignment submission patterns, participation in forums, and time spent on course materials.

One Potential Bias in the Data:

*Socioeconomic Bias:

Students from underprivileged backgrounds might be underrepresented in digital engagement data (e.g., fewer logins or submissions), leading to a model that unfairly associates low engagement with likelihood of dropping out, without considering external challenges like internet access or part-time work.

Preprocessing Steps:

**Handling Missing Data*:

Fill missing values in fields like attendance or GPA using appropriate imputation (e.g., mean GPA by major or semester).

**Normalization:*

Scale numeric features like GPA, hours logged in LMS, or number of credits taken using min-max scaling or z-score normalization to prepare for ML algorithms.

**Encoding Categorical Variables:*

Convert categorical variables (e.g., major, enrollment type, gender) into numerical format using one-hot encoding or label encoding.

3. Model Development

Model Choice: Neural Network

I would choose a Neural Network because of its ability to model complex, non-linear relationships in data. Neural networks are particularly effective when dealing with large datasets with many features, especially in image recognition, text analysis, or high-dimensional structured data. They can learn intricate patterns that simpler models might miss, leading to higher predictive accuracy if tuned and trained properly.

Data Splitting Strategy

To train and evaluate the model effectively, I would split the dataset as follows:

Training Set (70%): Used to train the neural network weights.

Validation Set (15%): Used during training to monitor performance and tune hyperparameters such as learning rate and layer size.

Test Set (15%): Used only at the end to assess the model's ability to generalize to new, unseen data.

If the dataset is small, I would use stratified k-fold cross-validation to ensure stability and efficient use of data.

Two Hyperparameters to Tune

* Learning Rate: Controls how much the model weights are updated during each training step. A value too high can overshoot minima; too low may slow training or get stuck. Tuning it is critical for convergence and performance.

* Number of Hidden Layers/Neurons: Affects model complexity and capacity. Too few may underfit, too many may overfit or be inefficient. Tuning this helps balance generalization and performance.

Neural Network Code Implementation

Codes

python

```
Import necessary libraries
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
```

```
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report
# Load and preprocess data (you can replace Iris with your own dataset)
data = load_iris()
X = data.data
y = data.target
# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
# One-hot encode target labels
y_encoded = tf.keras.utils.to_categorical(y, num_classes=3)
# Split data into train (70%), validation (15%), and test (15%)
X_train_val, X_test, y_train_val, y_test = train_test_split(X_scaled, y_encoded, test_size=0.15, random_state=42, stratify=y)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.176, random_state=42,
stratify=np.argmax(y_train_val, axis=1))
# Function to build a simple NN model with tunable parameters
def build_model(learning_rate=0.01, hidden_neurons=16):
    model = Sequential()
    model.add(Dense(hidden_neurons, input_dim=X.shape[1], activation='relu'))
    model.add(Dense(3, activation='softmax'))  # Output layer for 3 classes
    optimizer = Adam(learning_rate=learning_rate)
    model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    return model
# Instantiate and train the model
model = build_model(learning_rate=0.01, hidden_neurons=16)
history = model.fit(X_train, y_train, epochs=50, batch_size=8, validation_data=(X_val, y_val), verbose=0)

# Evaluate on test data
test_loss, test_accuracy = model.evaluate(X_test, y_test, verbose=0)
print(f"¥nTest Accuracy: {test_accuracy:.2f}")
# Classification report
y_pred = model.predict(X_test)
```

```
y_pred_classes = np.argmax(y_pred, axis=1)
y_true_classes = np.argmax(y_test, axis=1)
print("¥nClassification Report:")
print(classification_report(y_true_classes, y_pred_classes))
```

✅ Customizable Parameters:

```
learning_rate=0.01
hidden_neurons=16
```

You can tune these manually or use tools like KerasTuner for automation.

4. Evaluation & Deployment

Two Evaluation Metrics and Their Relevance

a. *Accuracy*

Accuracy is the ratio of correctly predicted instances to total predictions. It is useful when classes are balanced and gives a quick snapshot of model performance.

b. *F1-Score*

F1-Score is the harmonic mean of precision and recall. It's especially relevant in imbalanced datasets, where accuracy can be misleading. F1 balances false positives and false negatives, making it more informative.

codes

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, f1_score

# Load example data
data = load_iris()
X = data.data
y = data.target

# Split into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train a simple Neural Network
model = MLPClassifier(hidden_layer_sizes=(16,), max_iter=300, random_state=42)
model.fit(X_train, y_train)
```

```
# Make predictions
y_pred = model.predict(X_test)

# Evaluation
accuracy = accuracy_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred, average='weighted')

print(f"Accuracy: {accuracy:.2f}")
print(f"F1 Score: {f1:.2f}")
```

Concept drift

Concept drift refers to changes in the statistical properties of the target variable or input features over time, causing model performance to degrade after deployment.

Monitoring Methods:

Track real-time prediction accuracy or F1 score against new labeled data.

Use statistical drift detection algorithms (e.g., Population Stability Index, KL divergence).

Monitor model confidence (e.g., sharp drop in softmax probabilities).

Example: Detecting Drift with Model Confidence

Codes

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical
from sklearn.preprocessing import StandardScaler

# Load data
data = load_iris()
X = data.data
y = to_categorical(data.target)

# Standardize input
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

```
# Split into train and test (test will simulate new_X)
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

# Define and train a simple neural network
model = Sequential([
    Dense(16, activation='relu', input_shape=(X.shape[1],)),
    Dense(3, activation='softmax')
])
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(X_train, y_train, epochs=50, batch_size=8, verbose=0)

# Simulate new incoming data for concept drift check
new_X = X_test  # Replace with real-time data in production

# Predict and get confidence scores
confidences = np.max(model.predict(new_X), axis=1)
avg_confidence = np.mean(confidences)

print(f"Average model confidence on new data: {avg_confidence:.2f}")
if avg_confidence < 0.7:
    print("⚠ Warning: Possible concept drift detected.")
else:
    print("✅ No drift detected. Model confidence is stable.")
```

One technical challenge

Scalability

Scalability is the ability of the system to handle increasing amounts of load (e.g., users or data). For neural networks in production:

Serving many simultaneous requests may overload a single instance.

Model inference may slow down, leading to poor user experience.

Solution: Use model serving with batching or horizontal scaling using tools like TensorFlow Serving or FastAPI with Uvicorn + Gunicorn.

Example Deployment Snippet with FastAPI: