Machine Learning Intro | Assignment

Question 1: Explain the differences between AI, ML, Deep Learning (DL), and Data Science (DS).

Answer: AI is the broadest concept, ML is a subset of AI, Deep Learning is a subset of ML, and Data Science uses all three to extract insights from data.

Artificial Intelligence (AI)

- Definition: AI is the overarching field focused on creating machines that can simulate human intelligence.
- Goal: Enable machines to perform tasks like reasoning, problem-solving, understanding language, and perception.
- Examples: Chatbots, recommendation systems, autonomous vehicles, facial recognition.

Machine Learning (ML)

- Definition: A subset of AI that enables systems to learn from data and improve over time without being explicitly programmed. Goal: Develop algorithms that can identify patterns and make decisions based on data.
- Types:
- *Supervised Learning*: Learns from labeled data (e.g., spam detection).
- *Unsupervised Learning*: Finds patterns in unlabeled data (e.g., customer segmentation).
- *Reinforcement Learning*: Learns via trial and error (e.g., game-playing bots).

Deep Learning (DL)

- Definition: A specialized subset of ML that uses neural networks with many layers (hence "deep") to model complex patterns.
- Goal: Automatically learn hierarchical representations from data, especially unstructured data like images, audio, and text.
- Examples: Image classification, speech recognition, language translation.
- Tools: TensorFlow, PyTorch, Keras.

Data Science (DS)

- Definition: An interdisciplinary field that uses scientific methods, algorithms, and systems to extract insights from structured and unstructured data.
- Goal: Turn raw data into actionable insights for decision-making.
- Includes:
    - Data collection and cleaning
    - Exploratory data analysis
    - Statistical modeling
    - Visualization
    - Use of AI/ML for predictive analytics
- Tools: Python (Pandas, NumPy), SQL, Power BI, R, Jupyter, etc.

Question 2: What are the types of machine learning? Describe each with one real-world example.

Answer: Machine learning is typically categorized into three main types: Supervised, Unsupervised, and Reinforcement Learning. Each type has a distinct learning approach and real-world applications.

1. Supervised Learning

- Definition: The model learns from a labeled dataset, where each input has a corresponding correct output.
- Goal: Predict outcomes for new, unseen data based on past labeled data.
- Common Algorithms: Linear Regression, Decision Trees, Support Vector Machines (SVM), Neural Networks.
- Real-World Example:
  Email Spam Detection – The model is trained on emails labeled as "spam" or "not spam" and learns to classify new emails accordingly**.**

2. Unsupervised Learning

- Definition: The model works with unlabeled data and tries to find hidden patterns or groupings.
- Goal: Discover structure or relationships in data without predefined labels.
- Common Algorithms: K-Means Clustering, Hierarchical Clustering, Principal Component Analysis (PCA).
- Real-World Example:
  Customer Segmentation – E-commerce platforms use unsupervised learning to group customers based on purchasing behavior to tailor marketing strategies.

3. Reinforcement Learning

- Definition: The model learns by interacting with an environment and receiving feedback in the form of rewards or penalties.
- Goal: Learn a sequence of actions that maximize cumulative reward.
- Common Algorithms: Q-Learning, Deep Q Networks (DQN), Policy Gradient Methods.
- Real-World Example:
  Self-Driving Cars – The system learns to navigate by receiving rewards for safe driving and penalties for mistakes like collisions or traffic violations.

Semi-Supervised & Self-Supervised Learning

- **Semi-Supervised**: Combines a small amount of labeled data with a large amount of unlabeled data. Used when labeling is expensive.
- **Self-Supervised**: A form of unsupervised learning where the system generates its own labels from the data (e.g., predicting the next word in a sentence).

Question 3: Define overfitting, underfitting, and the bias-variance tradeoff in machine learning.

Answer: Overfitting occurs when a model learns noise instead of patterns, underfitting happens when it fails to learn enough, and the bias-variance tradeoff is the balance between these two extremes to achieve optimal model performance.

Overfitting

- Definition: A model is said to overfit when it learns the training data *too well*, including its noise and outliers, which harms its performance on new, unseen data.
- Symptoms:
  - Very high accuracy on training data
  - Poor performance on test/validation data
- Cause: Model is too complex (e.g., too many parameters or layers) relative to the amount of training data.
- Example: A decision tree that grows too deep and memorizes every training point, including anomalies, but fails to generalize to new data.

Underfitting

- Definition: A model underfits when it is too simple to capture the underlying structure of the data.
- Symptoms:
  - Poor performance on both training and test data
- Cause: Model is too basic (e.g., linear regression on a nonlinear problem), or not trained long enough.
- Example: Using a linear model to predict complex housing prices where relationships between features and price are nonlinear.

Bias-Variance Tradeoff

- Definition: This is the fundamental tension between two sources of error in machine learning models:
  - Bias: Error due to overly simplistic assumptions in the model. High bias leads to underfitting.
  - Variance: Error due to model sensitivity to small fluctuations in the training set. High variance leads to overfitting.
- Goal: Find the sweet spot where both bias and variance are minimized enough to ensure good generalization.
- Visualization:
- High Bias → Simple model → Misses patterns → Underfitting
- High Variance → Complex model → Captures noise → Overfitting
- Balanced Bias & Variance → Optimal model performance.

Question 4: What are outliers in a dataset, and list three common techniques for handling them.

Answer: Outliers are data points that significantly deviate from the rest of the dataset, potentially indicating variability, errors, or rare events. Handling them is crucial because they can distort statistical analyses and machine learning models.

What Are Outliers?

- Definition: An outlier is an observation that lies an abnormal distance from other values in a dataset.
- Causes:
  - Data entry or measurement errors
  - Natural variation in data
  - Experimental errors or rare events
- Impact:
- Skewed statistical summaries (mean, standard deviation)
- Poor model performance due to overfitting or misleading patterns

Common Techniques to Handle Outliers

1. **Z-Score Method**
   - How it works: Calculates how many standard deviations a data point is from the mean.
   - Rule of thumb: If Z > 3, the point is considered an outlier.
   - Use case: Effective for normally distributed data.
   - Example:

```
from scipy import stats

z_scores = stats.zscore(data)

outliers = np.where(np.abs(z_scores) > 3)
```

2. **Interquartile Range (IQR) Method**
   - How it works: Measures the spread of the middle 50% of data (Q3 - Q1).
   - Rule: Outliers lie below Q1 - 1.5×IQR or above Q3 + 1.5×IQR.
   - Use case: Robust for skewed distributions.
   - Example:

```
Q1 = data.quantile(0.25)

Q3 = data.quantile(0.75)

IQR = Q3 - Q1

outliers = data[(data < Q1 - 1.5 * IQR) | (data > Q3 + 1.5 * IQR)]
```

3. **Isolation Forest**

- How it works: An ensemble algorithm that isolates anomalies by randomly selecting features and split values.

- Use case: Works well with high-dimensional datasets.
- **Example:**

```
from sklearn.ensemble import IsolationForest

model = IsolationForest()

model.fit(data)

predictions = model.predict(data)  # -1 for outliers, 1 for inliers
```

When to Remove or Keep Outliers

- Remove: If they are due to errors or irrelevant to the analysis.
- Keep: If they represent rare but valid phenomena (e.g., fraud detection, rare diseases).

Question 5: Explain the process of handling missing values and mention one imputation technique for numerical and one for categorical data.

Answer: Handling missing values is a crucial step in data preprocessing, as they can distort analysis and model performance. Here's a structured approach.

Process of Handling Missing Values

1. **Identify Missing Data**
    - Use methods like `.isnull()`, `.info()`, or `.describe()` in Pandas to detect missing values.
    - Visualize missingness using libraries like `missingno` or `seaborn`.
2. **Understand the Pattern**
    - Determine if data is:
    - MCAR (Missing Completely at Random)
    - MAR (Missing at Random)
    - MNAR (Missing Not at Random)
3. **Decide on a Strategy**
    - Remove: Drop rows or columns with too many missing values.
    - Impute: Fill in missing values using statistical or model-based methods.
    - Flag: Create an indicator variable to mark missingness (useful for tree-based models).
4. **Apply the Chosen Method**
    - Use appropriate imputation techniques based on data type and context.
5. **Validate**
    - Check the impact of imputation on data distribution and model performance.

Imputation Technique for Numerical Data: Mean Imputation

- What it does: Replaces missing values with the mean of the non-missing values in the column.
- When to use: When data is normally distributed and missingness is MCAR.

- Example:

  df['age'].fillna(df['age'].mean(), inplace=True)

Imputation Technique for Categorical Data: Mode Imputation

- What it does: Replaces missing values with the most frequent category (mode).
- When to use: When a dominant category exists and missingness is not informative.
- Example:

  df['gender'].fillna(df['gender'].mode()[0], inplace=True)

These are simple yet effective techniques, especially for baseline models. For more advanced handling, you can explore:

- KNN Imputation
- Multivariate Imputation by Chained Equations (MICE)
- Model-based Imputation (e.g., using regression)

Question 6: Write a Python program that:

● Creates a synthetic imbalanced dataset with make_classification() from sklearn.datasets.

● Prints the class distribution.

Answer:

Python program that creates a synthetic imbalanced dataset using `make_classification()` from `sklearn.datasets` and prints the class distribution using `collections.Counter`:

```
from sklearn.datasets import make_classification

from collections import Counter

import numpy as np

# Create synthetic imbalanced dataset

X, y = make_classification(n_samples=1000, n_features=20, n_informative=2,

                          n_redundant=10, n_clusters_per_class=1,

                          weights=[0.9, 0.1], flip_y=0, random_state=42)

# Print class distribution

class_counts = Counter(y)

print("Class distribution:")

for label, count in class_counts.items():
    print(f"Class {label}: {count} samples ({(count/len(y))*100:.2f}%)")
```

Explanation:

- `weights=[0.9, 0.1]` creates a 90:10 imbalance between the two classes.
- `Counter(y)` counts how many samples belong to each class.
- You can adjust `n_samples`, `weights`, or `random_state` to experiment with different imbalances.

Want to visualize this imbalance with a plot using Seaborn or Matplotlib? I can help you add that too.

Question 7: Implement one-hot encoding using pandas for the following list of colors: ['Red', 'Green', 'Blue', 'Green', 'Red'].

Print the resulting dataframe.

Answer: Python program that performs one-hot encoding using Pandas for the given list of colors:

```
import pandas as pd


# Original list of colors

colors = ['Red', 'Green', 'Blue', 'Green', 'Red']

# Convert to DataFrame

df = pd.DataFrame({'Color': colors})

# Apply one-hot encoding

encoded_df = pd.get_dummies(df, columns=['Color'])


# Print the resulting DataFrame

print(encoded_df)
```

Output:

This will produce a DataFrame like:

```
Color_Blue  Color_Green  Color_Red

0        0          0        1

1        0          1        0

2        1          0        0
```

3	0	1	0
4	0	0	1

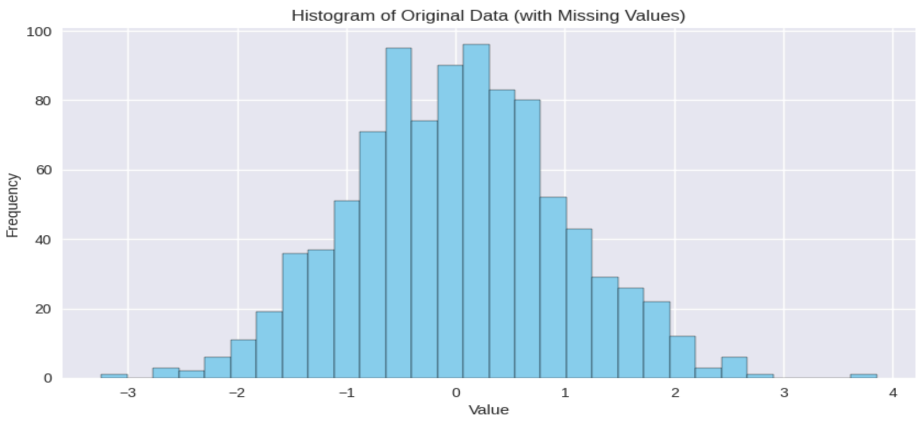
Question 8: Write a Python script to:

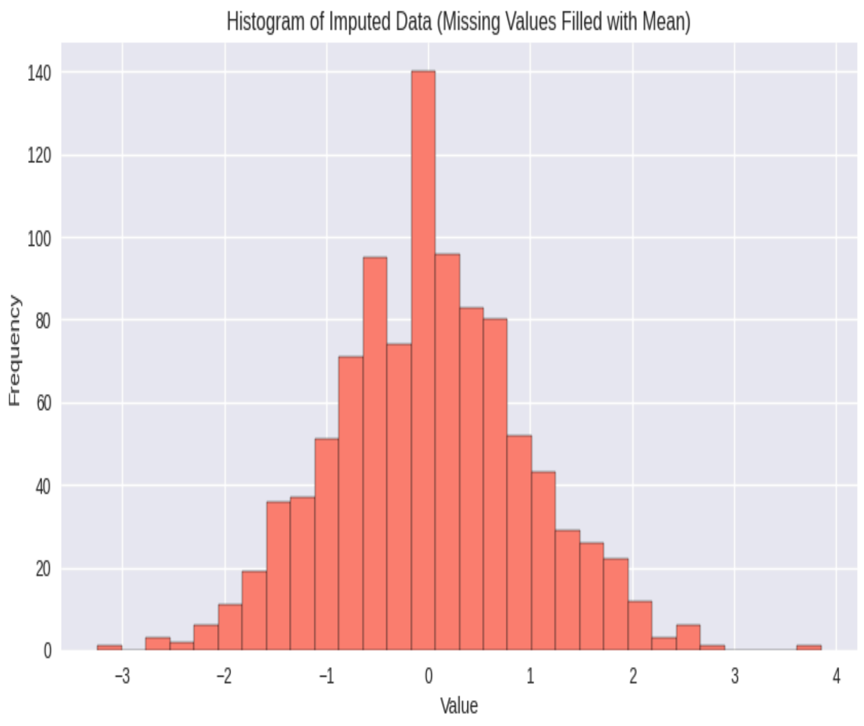
- **Generate 1000 samples from a normal distribution.**
- **Introduce 50 random missing values.**
- **Fill missing values with the column mean.**
- **Plot a histogram before and after imputation.**

Answer: Python workflow that generates 1000 normal samples, introduces 50 missing values, fills them with the mean, and visualizes the data before and after imputation.

```
# Generating and visualizing histograms before and after imputing missing values in a normal distribution

import numpy as np

import matplotlib.pyplot as plt

import os

# Ensure output directory exists

os.makedirs("/mnt/data", exist_ok=True)

# Step 1: Generate 1000 samples from a normal distribution

np.random.seed(42)

data = np.random.normal(loc=0, scale=1, size=1000)

# Step 2: Introduce 50 random missing values

missing_indices = np.random.choice(len(data), size=50, replace=False)

data_with_nans = data.copy()

data_with_nans[missing_indices] = np.nan

# Step 3: Plot histogram of original data with missing values

plt.style.use('seaborn-v0_8')

plt.figure(figsize=(10, 5))

plt.hist(data_with_nans[~np.isnan(data_with_nans)], bins=30, color='skyblue', edgecolor='black')

plt.title('Histogram of Original Data (with Missing Values)')

plt.xlabel('Value')

plt.ylabel('Frequency')

plt.grid(True)
```

```
original_hist_path = "/mnt/data/original_histogram.png"

plt.savefig(original_hist_path)

plt.close()

# Step 4: Fill missing values with column mean

mean_value = np.nanmean(data_with_nans)

data_imputed = np.where(np.isnan(data_with_nans), mean_value, data_with_nans)

# Step 5: Plot histogram of imputed data

plt.figure(figsize=(10, 5))

plt.hist(data_imputed, bins=30, color='salmon', edgecolor='black')

plt.title('Histogram of Imputed Data (Missing Values Filled with Mean)')

plt.xlabel('Value')

plt.ylabel('Frequency')

plt.grid(True)

imputed_hist_path = "/mnt/data/imputed_histogram.png"

plt.savefig(imputed_hist_path)

plt.close()

print("Generated 1000 normal samples, introduced 50 missing values, plotted original and imputed histograms as
'original_histogram.png' and 'imputed_histogram.png'.")
```





Breakdown of the Process

- Step 1: Data Generation
  - Created 1000 samples from a standard normal distribution using `np.random.normal()`.
- Step 2: Introduced Missing Values
  - Randomly selected 50 indices and replaced them with `np.nan`.
- Step 3: Imputation
  - Calculated the mean of the non-missing values and filled the missing entries using `np.nanmean()`.
- Step 4: Visualization
- Plotted two histograms:
- One excluding missing values (original distribution).
- One after mean imputation (filled distribution).

- **Before Imputation**: The histogram reflects the original distribution but excludes the 50 missing values.
- **After Imputation**: The imputed values (equal to the mean) slightly increase the frequency around the mean, making the central peak more pronounced.

Question 9: Implement Min-Max scaling on the following list of numbers [2, 5, 10, 15, 20] using sklearn.preprocessing.MinMaxScaler. Print the scaled array.

Answer: Python script that uses `MinMaxScaler` from `sklearn.preprocessing` to scale the list `[2, 5, 10, 15, 20]`:

```
from sklearn.preprocessing import MinMaxScaler
import numpy as np

# Original data
data = np.array([[2], [5], [10], [15], [20]])

# Initialize the scaler
scaler = MinMaxScaler()

# Fit and transform the data
scaled_data = scaler.fit_transform(data)

# Print the scaled array
print("Scaled Data:")
print(scaled_data)
```

Output:

The scaled values will be between 0 and 1:

```
[[0.        ]

 [0.15789474]

 [0.42105263]

 [0.68421053]

 [1.        ]]
```

Question 10: You are working as a data scientist for a retail company. You receive a customer transaction dataset that contains:

- **Missing ages,**
- **Outliers in transaction amount,**
- **A highly imbalanced target (fraud vs. non-fraud),**
- **Categorical variables like payment method.**

Explain the step-by-step data preparation plan you'd follow before training a machine learning model. Include how you'd address missing data, outliers, imbalance, and encoding.

Answer:

1. **Exploratory Data Analysis (EDA)**

- Inspect dataset structure: Use `.info()`, `.describe()`, and `.head()` to understand data types and distributions.
- Visualize distributions: Histograms, boxplots, and count plots to spot missing values, outliers, and imbalance.
- Check correlations: Use heatmaps or `.corr()` to identify relationships between features and the target.

2. Handling Missing Data (e.g., Missing Ages)

- Assess missingness:
    - Use `.isnull().sum()` and `missingno.matrix()` to visualize.
    - Determine if missingness is MCAR, MAR, or MNAR.
- Imputation strategy:
- Numerical (Age): Use *mean*, *median*, or *KNN imputation* depending on distribution.

df['age'].fillna(df['age'].mean(), inplace=True)

Optionally, create a binary flag column:

```
age_missing = df['age'].isnull().astype(int)
```

3. Handling Outliers (e.g., Transaction Amount)

- Detect outliers:
    - Use IQR method or Z-score.
    - Visualize with boxplots or log-transformed histograms.
- Treatment options:
- Cap/Floor: Winsorize extreme values.
- Log Transformation: Reduce skewness.

df['transaction_amount'] = np.log1p(df['transaction_amount'])

Remove: If clearly erroneous and not fraud-related.

4. Addressing Class Imbalance (Fraud vs. Non-Fraud)

Check imbalance:

df['target'].value_counts(normalize=True)

Resampling techniques:

- Oversampling: SMOTE, ADASYN
- Undersampling: Random undersampling of majority class
- Hybrid: Combine both

Alternative strategies:

Use class weights in models like Logistic Regression or XGBoost:

model = LogisticRegression(class_weight='balanced')

5. Encoding Categorical Variables (e.g., Payment Method)

Low cardinality: Use One-Hot Encoding

```
pd.get_dummies(df['payment_method'], drop_first=True)
```

- High cardinality: Use Target Encoding or Frequency Encoding
- Ordinal features: Use Ordinal Encoding if order matters

6. Feature Scaling

- Apply scaling to numerical features:
- Use Min-Max Scaling or Standardization depending on algorithm

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

df[['amount', 'age']] = scaler.fit_transform(df[['amount', 'age']])
```

7. Train-Test Split & Validation Strategy

- **Stratified split** to preserve class distribution

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, test_size=0.2)
```

Use **cross-validation** with stratification for robust evaluation

8. Model Training & Evaluation

- Try models like:
  - Logistic Regression (with class weights)
  - Random Forest
  - XGBoost (handles imbalance well)
- Use metrics suited for imbalance:
- Precision, Recall, F1-score
- ROC-AUC
- Confusion Matrix