☑ 1. Data Preprocessing

Data preprocessing is essential to clean and prepare data before feeding it into a machine learning model.

Steps:

- Handling missing values
- Encoding categorical variables
- Feature scaling

Example:

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder, StandardScaler from
sklearn.impute import SimpleImputer

# Sample dataset data
= pd.DataFrame({
   'Age': [25, 27, 29, None, 22],
   'Gender': ['Male', 'Female', 'Female', 'Male', 'Female'],
   'Salary': [50000, 54000, 58000, 60000, None]
})

# Handle missing values
imputer = SimpleImputer(strategy='mean')
data[['Age', 'Salary']] = imputer.fit_transform(data[['Age', 'Salary']])

# Encode categorical variable le
= LabelEncoder()
data['Gender'] = le.fit_transform(data['Gender'])  # Male=1, Female=0

# Feature Scaling scaler
= StandardScaler()
data[['Age', 'Salary']] = scaler.fit_transform(data[['Age', 'Salary']])

print(data)
```

Import the tools:

- import pandas as pd - Brings in the tool for working with data tables
- from sklearn.preprocessing import LabelEncoder, StandardScaler - Tools for converting text to numbers and scaling data
- from sklearn.impute import SimpleImputer - Tool for filling in missing values

Create sample data:

- data = pd.DataFrame({...}) - Creates a table with Age, Gender, and Salary columns, including some missing values (None)

Handle missing values:

- imputer = SimpleImputer(strategy='mean') - Creates a tool that fills missing values with the average
- data[['Age', 'Salary']] = imputer.fit_transform(data[['Age', 'Salary']]) - Replaces None values with the mean of each column

Convert text to numbers:

- le = LabelEncoder() - Creates a tool that converts text categories to numbers
- data['Gender'] = le.fit_transform(data['Gender']) - Changes 'Male' to 1 and 'Female' to 0

Scale the numbers:

- scaler = StandardScaler() - Creates a tool that makes all numbers have similar ranges
- data[['Age', 'Salary']] = scaler.fit_transform(data[['Age', 'Salary']]) - Converts Age and Salary to standardized values (mean=0, std=1)

Show results:

- print(data) - Displays the cleaned, processed data ready for machine learning

This preprocessing ensures the data has no missing values, all categories are numeric, and all features are on the same scale for better model performance.

☑ 2. Model Selection and Evaluation

Choose the best model using **training/test split**, **cross-validation**, and evaluate using metrics like **accuracy**, **precision**, **recall**, **F1-score**.

Example (Classification):

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report from
sklearn.ensemble import RandomForestClassifier

# Features and labels
X = data[['Age', 'Salary', 'Gender']]
y = [1, 0, 0, 1, 0]  # Sample target (binary classification)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Model
model = RandomForestClassifier() model.fit(X_train,
y_train)

# Predictions
y_pred = model.predict(X_test)

# Evaluation
print("Accuracy:", accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

Here's a step-by-step explanation of the complete classification workflow:

Import the tools:

- from sklearn.model_selection import train_test_split - Tool to split data into training and testing sets
- from sklearn.metrics import accuracy_score, classification_report - Tools to measure how well the model performed
- from sklearn.ensemble import RandomForestClassifier - A powerful classifier that uses multiple decision trees

Prepare the data:

- X = data[['Age', 'Salary', 'Gender']] - Features (input variables) we'll use to make predictions
- y = [1, 0, 0, 1, 0] - Target labels (what we want to predict: 1 or 0)

Split the data:

- X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42) - Separates 80% for training and 20% for testing
- The number 42 in random_state=42 is just an arbitrary choice - it could be any number!

Train the model:

- model = RandomForestClassifier() - Creates a random forest model
- model.fit(X_train, y_train) - Teaches the model using the training data

Make predictions:

- y_pred = model.predict(X_test) - Uses the trained model to predict labels for the test data

Evaluate performance:

- print("Accuracy:", accuracy_score(y_test, y_pred)) - Shows what percentage of predictions were correct
- print(classification_report(y_test, y_pred)) - Shows detailed performance metrics like precision and recall

This workflow ensures we test the model on data it hasn't seen before, giving us a realistic measure of how well it will perform on new data.

☑ 3. Supervised Learning Algorithms

Regression Example (Linear Regression):

```
from sklearn.linear_model import LinearRegression import
numpy as np

# Dataset
X = np.array([[1], [2], [3], [4], [5]])
y = np.array([2, 4, 6, 8, 10])

model = LinearRegression()
model.fit(X, y)

# Predict
print(model.predict([[6]]))  # Output: 12
```

Here's a step-by-step explanation of the linear regression code:

Import libraries:

- from sklearn.linear_model import LinearRegression - Brings in the tool that creates straight-line predictions

- import numpy as np - Brings in a library for handling numbers and arrays

Create the dataset:

- X = np.array([[1], [2], [3], [4], [5]]) - These are our input values (what we know)

- y = np.array([2, 4, 6, 8, 10]) - These are our output values (what we want to predict)

Train the model:

- model = LinearRegression() - Creates an empty linear regression model

- model.fit(X, y) - Teaches the model by showing it the relationship between X and y

Make a prediction:

- print(model.predict([[6]])) - Asks the model "if X is 6, what should y be?"

- # Output: 12 - The model predicts 12 because it learned the pattern ($y = 2 \times X$)

The model figured out that every input number gets doubled to make the output, so when you give it 6, it predicts 12.

Classification Example (Logistic Regression):

```
from sklearn.linear_model import LogisticRegression

# Sample binary classification
X = [[0], [1], [2], [3]] y
= [0, 0, 1, 1]

model = LogisticRegression()
model.fit(X, y)

print(model.predict([[1.5]]))  # Output: [0] or [1]
```

Here's a step-by-step explanation of the logistic regression code:

Import the tool:

- from sklearn.linear_model import LogisticRegression - Brings in the tool that makes yes/no predictions

Create the dataset:

- X = [[0], [1], [2], [3]] - These are our input values (features we measure)
- y = [0, 0, 1, 1] - These are our labels (0 means "no", 1 means "yes")

Train the model:

- model = LogisticRegression() - Creates an empty logistic regression model
- model.fit(X, y) - Teaches the model using the input-output pairs

Make a prediction:

- print(model.predict([[1.5]])) - Asks the model "for input 1.5, is it class 0 or 1?"
- # Output: [0] or [1] - The model predicts either 0 or 1 based on what it learned

The model learned that smaller numbers (0,1) go with class 0, and larger numbers (2,3) go with class 1, so 1.5 falls somewhere in between and the model decides which side of the boundary it's on.

☑ 4. Unsupervised Learning Algorithms

Clustering Example (KMeans):

```
from sklearn.cluster import KMeans

data = [[1, 2], [1, 4], [1, 0],
[10, 2], [10, 4], [10, 0]]

kmeans = KMeans(n_clusters=2)
kmeans.fit(data)

print("Cluster Centers:", kmeans.cluster_centers_) print("Labels:",
kmeans.labels_)
```

Here's a step-by-step explanation of the KMeans clustering code:

Import the tool:

- from sklearn.cluster import KMeans - Brings in the tool that groups similar data points together

Create the dataset:

- data = [[1, 2], [1, 4], [1, 0], [10, 2], [10, 4], [10, 0]] - Six data points with x,y coordinates

Set up the clustering:

- kmeans = KMeans(n_clusters=2) - Creates a KMeans model that will find 2 groups
- kmeans.fit(data) - Analyzes the data to find the best way to split it into 2 clusters

Show the results:

- print("Cluster Centers:", kmeans.cluster_centers_) - Shows the center point of each cluster
- print("Labels:", kmeans.labels_) - Shows which cluster each original data point belongs to

The algorithm found two groups: points around x=1 (left side) and points around x=10 (right side), then calculated the average center of each group and labeled each point as belonging to cluster 0 or 1.

Dimensionality Reduction (PCA):

```
from sklearn.decomposition import PCA import
numpy as np

# Sample 3D data
X = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

pca = PCA(n_components=2) X_reduced
= pca.fit_transform(X)

print("Reduced Data:\n", X_reduced)
```

Here's a step-by-step explanation of the PCA dimensionality reduction code:

Import the tools:

- from sklearn.decomposition import PCA - Brings in the tool that reduces data dimensions
- import numpy as np - Brings in the library for handling arrays of numbers

Create the dataset:

- X = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]) - Three data points, each with 3 dimensions (x, y, z coordinates)

Set up dimension reduction:

- pca = PCA(n_components=2) - Creates a PCA model that will reduce data from 3D to 2D
- X_reduced = pca.fit_transform(X) - Analyzes the 3D data and converts it to 2D while keeping the most important information

Show the results:

- print("Reduced Data:\n", X_reduced) - Displays the new 2D version of the original 3D data

PCA found the best way to squash 3D data onto a 2D plane by keeping the directions where the data varies the most, like taking a shadow of a 3D object that preserves its most important features.

5. Pipelines

Pipelines help streamline the preprocessing and modeling steps.

Example:

```
from sklearn.pipeline import Pipeline from
sklearn.preprocessing import StandardScaler from
sklearn.linear_model import LogisticRegression

# Sample dataset
X = [[0.5], [2.5], [1.0], [3.5]] y
= [0, 1, 0, 1]

pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', LogisticRegression())
])

pipeline.fit(X, y)
print(pipeline.predict([[1.5]]))
```

Here's a step-by-step explanation of the pipeline code:

Import the tools:

- from sklearn.pipeline import Pipeline - Brings in the tool that chains multiple steps together
- from sklearn.preprocessing import StandardScaler - Brings in the tool that normalizes data values
- from sklearn.linear_model import LogisticRegression - Brings in the classification tool

Create the dataset:

- X = [[0.5], [2.5], [1.0], [3.5]] - Input values with different scales
- y = [0, 1, 0, 1] - Corresponding class labels (0 or 1)

Build the pipeline:

- pipeline = Pipeline([('scaler', StandardScaler()), ('classifier', LogisticRegression())]) - Creates a two-step process: first normalize data, then classify

Train and predict:

- pipeline.fit(X, y) - Runs both steps: scales the training data, then trains the classifier
- print(pipeline.predict([[1.5]])) - Automatically scales the new input [1.5] then makes a prediction

The pipeline ensures that any new data gets the same preprocessing (scaling) as the training data before making predictions, which is crucial for consistent results.