Data preprocessing:

Certainly! Here’s a Python cheat sheet for the essential functions used in data preprocessing for machine learning, including the libraries needed and examples for reading and storing a .csv file. I’ll cover each of the topics you mentioned.

**Libraries Required**

import pandas as pd

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

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

from sklearn.preprocessing import StandardScaler, MinMaxScaler, OneHotEncoder, LabelEncoder

**1. Handling Missing Data**

pythonCopy code

# Load data

df = pd.read\_csv('data.csv')

# Check missing values

df.isnull().sum()

# Drop rows with missing values

df.dropna(inplace=True)

# Fill missing values with mean of the column

df.fillna(df.mean(), inplace=True)

# Numerical columns: impute with mean

numerical\_cols = df.select\_dtypes(include=['int64', 'float64']).columns

# Categorical columns: impute with most frequent

categorical\_cols = df.select\_dtypes(include=['object']).columns

# Impute missing values

for col in numerical\_cols:

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

**2. Formatting Date Columns**

# Convert a column to datetime

df['date\_column'] = pd.to\_datetime(df['date\_column'])

# Extract year, month, day, etc.

df['year'] = df['date\_column'].dt.year

df['month'] = df['date\_column'].dt.month

df['day'] = df['date\_column'].dt.day

**3. Visualising Dataset Before and After Handling Missing Values**

# Before

sns.heatmap(df.isnull(), cbar=False)

plt.show()

# After handling missing data

df.fillna(df.mean(), inplace=True)

sns.heatmap(df.isnull(), cbar=False)

plt.show()

**4. Handling Outliers Using IQR**

# Calculate IQR

Q1 = df['column'].quantile(0.25)

Q3 = df['column'].quantile(0.75)

IQR = Q3 - Q1

# Filter out outliers

df = df[(df['column'] >= (Q1 - 1.5 \* IQR)) & (df['column'] <= (Q3 + 1.5 \* IQR))]

for multiple columns:

for col in numerical\_cols:

    if col != 'type':  # 'type' is now encoded, no longer numerical

        Q1 = df[col].quantile(0.25)

        Q3 = df[col].quantile(0.75)

        IQR = Q3 - Q1

        lower\_bound = Q1 - 1.5 \* IQR

        upper\_bound = Q3 + 1.5 \* IQR

        df[col] = df[col].clip(lower=lower\_bound, upper=upper\_bound)

**5. Visualising Outliers Before and After Preprocessing**

# Before

sns.boxplot(x=df['column'])

plt.show()

# After removing outliers

filtered\_df = df[(df['column'] >= (Q1 - 1.5 \* IQR)) & (df['column'] <= (Q3 + 1.5 \* IQR))]

sns.boxplot(x=filtered\_df['column'])

plt.show()

**6. Categorical Encoding - One Hot and Label**

# One Hot Encoding

df\_encoded = pd.get\_dummies(df, columns=['categorical\_column'])

# Label Encoding

label\_encoder = LabelEncoder()

df['categorical\_column'] = label\_encoder.fit\_transform(df['categorical\_column'])

**7. Feature Scaling - Standardisation, Normalisation**

# Standardisation

scaler = StandardScaler()

df['scaled\_column'] = scaler.fit\_transform(df[['numeric\_column']])

# Normalisation

min\_max\_scaler = MinMaxScaler()

df['normalized\_column'] = min\_max\_scaler.fit\_transform(df[['numeric\_column']])

**8. Splitting the Dataset into Test and Train**

# Split dataset

X\_train, X\_test, y\_train, y\_test = train\_test\_split(df.drop('target\_column', axis=1), df['target\_column'], test\_size=0.2, random\_state=42)

**Reading and Storing a .csv File**

# Reading a CSV file

df = pd.read\_csv('data.csv')

# Storing a DataFrame as a CSV file

df.to\_csv('processed\_data.csv', index=False)

Regression:

**Libraries Required**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

import statsmodels.api as sm

from sklearn.linear\_model import LinearRegression

**1. Produce Scatterplots for Each Independent Variable Against the Dependent Variable (Price)**

# Assuming 'price' is the dependent variable and others are independent variables

independent\_vars = df.columns.drop('price')  # Adjust based on your dataset

for var in independent\_vars:

    sns.scatterplot(x=df[var], y=df['price'])

    plt.title(f'Scatterplot of {var} vs Price')

    plt.xlabel(var)

    plt.ylabel('Price')

    plt.show()

**2. Interpret the Strength of the Relationship Via the Correlation Matrix**

# Calculate correlation matrix

correlation\_matrix = df.corr()

# Plot heatmap of correlation matrix

sns.heatmap(correlation\_matrix, annot=True, cmap='coolwarm')

plt.title('Correlation Matrix')

plt.show()

**3. Fit a Linear Model to the Data to Predict the Prices of Avocados Using Linear Regression**

lin\_reg = LinearRegression()

lin\_reg.fit(X\_train, y\_train)

from sklearn.metrics import mean\_squared\_error, r2\_score

y\_pred\_train = lin\_reg.predict(X\_train)

y\_pred\_test = lin\_reg.predict(X\_test)

print('Train R-squared:', r2\_score(y\_train, y\_pred\_train))

print('Test R-squared:', r2\_score(y\_test, y\_pred\_test))

print('Train MSE:', mean\_squared\_error(y\_train, y\_pred\_train))

print('Test MSE:', mean\_squared\_error(y\_test, y\_pred\_test))

**4. Assess the Strength of the Relationship Through statsmodels.summary()**

import statsmodels.api as sm

X\_train\_sm = sm.add\_constant(X\_train)

model = sm.OLS(y\_train, X\_train\_sm).fit()

print(model.summary())

**5. Based on the Scatterplots and Correlation Values, Identify Which Variable(s) You Will Use to Predict the Price of Avocados**

# Method 2: Correlation Matrix Analysis

correlation\_matrix = train\_df.corr()

correlation\_with\_target = correlation\_matrix['AveragePrice'].sort\_values(ascending=False)

correlation\_with\_target = correlation\_with\_target.drop('AveragePrice')

print("Correlation matrix:\n", correlation\_with\_target)

# Determine thresholds based on percentiles

# For positive correlations

positive\_threshold = correlation\_with\_target[correlation\_with\_target > 0].quantile(0.75)

# For negative correlations

negative\_threshold = correlation\_with\_target[correlation\_with\_target < 0].quantile(0.25)

# Select variables that are highly positively and negatively correlated based on these thresholds

highly\_positively\_correlated\_vars = correlation\_with\_target[correlation\_with\_target > positive\_threshold].index.tolist()

highly\_negatively\_correlated\_vars = correlation\_with\_target[correlation\_with\_target < negative\_threshold].index.tolist()

# Output the selected variables

print("Highly positively correlated variables:", highly\_positively\_correlated\_vars)

print("Highly negatively correlated variables:", highly\_negatively\_correlated\_vars)

import statsmodels.api as sm

# Assuming X\_train and y\_train are already defined and preprocessed

X\_train\_sm = sm.add\_constant(X\_train)  # Adding a constant for the intercept

model = sm.OLS(y\_train, X\_train\_sm).fit()

# Select significant variables based on p-value < 0.05

significant\_vars = list(model.pvalues[model.pvalues < 0.01].index)

if 'const' in significant\_vars:

    significant\_vars.remove('const')  # Remove the constant

# Print the names of the significant variables and their p-values

print("Significant variables and their p-values:")

for var in significant\_vars:

    print(f"{var}: p-value={model.pvalues[var]}")

# Combine lists from correlation analysis and OLS summary

final\_vars = list(set(highly\_positively\_correlated\_vars + highly\_negatively\_correlated\_vars + significant\_vars ))

print(final\_vars)

**6. Use the Appropriate Independent Variables and Fit a Linear Model**

# Fit a linear model using only the selected variables

X\_train\_final = X\_train[final\_vars]

X\_test\_final = X\_test[final\_vars]

lin\_reg\_final = LinearRegression()

lin\_reg\_final.fit(X\_train\_final, y\_train)

# Predict and evaluate the final model using test data

y\_pred\_final = lin\_reg\_final.predict(X\_test\_final)

# Calculate metrics for the new model

mse\_final = mean\_squared\_error(y\_test, y\_pred\_final)

r2\_final = r2\_score(y\_test, y\_pred\_final)

print(f'New Final model MSE: {mse\_final}')

print(f'New Final model R-squared: {r2\_final}')

**Visualize the Results**

# Create a scatter plot of actual vs. predicted values

plt.figure(figsize=(10, 6))

plt.scatter(y\_test, y\_pred\_final, alpha=0.3)

# Plot a line of perfect prediction

min\_val = min(y\_test.min(), y\_pred\_final.min())

max\_val = max(y\_test.max(), y\_pred\_final.max())

plt.plot([min\_val, max\_val], [min\_val, max\_val], color='red', linestyle='--', lw=2)

plt.title('Actual vs. Predicted Prices')

plt.xlabel('Actual Prices')

plt.ylabel('Predicted Prices')

plt.show()

**LOGISTIC REGRESSION:**

**Fit model and confusion matrix**

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import confusion\_matrix, classification\_report

# Creating a pipeline that first scales the data then applies logistic regression

pipeline = Pipeline([

    ('scaler', StandardScaler()),

    ('logistic', LogisticRegression(solver='liblinear', max\_iter=1000))

])

# Fitting the model using the pipeline. The pipeline will first transform the data using StandardScaler and then fit the logistic regression model.

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

# You can access the logistic regression model directly via pipeline.named\_steps['logistic']

logistic\_model = pipeline.named\_steps['logistic']

# The model is now fitted and can be used to make predictions on the scaled test data automatically

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

# Generating the confusion matrix

cm = confusion\_matrix(y\_test, y\_pred)

# Visualizing the confusion matrix

plt.figure(figsize=(8, 6))

sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=['Predicted 0', 'Predicted 1'], yticklabels=['Actual 0', 'Actual 1'])

plt.title('Confusion Matrix')

plt.ylabel('Actual label')

plt.xlabel('Predicted label')

plt.show()

**precision, recall and f1**

# Classification Report for precision, recall, and F1 score

print(classification\_report(y\_test, y\_pred, target\_names=['No Rain', 'Rain']))

# Calculating and Displaying Specificity and Classification Error

tn, fp, fn, tp = cm.ravel()

specificity = tn / (tn + fp)

classification\_error = (fp + fn) / (tp + tn + fp + fn)

print(f"Specificity: {specificity:.2f}")

print(f"Classification Error: {classification\_error:.2f}")

**ROC curve:**

from sklearn.metrics import roc\_curve, auc

import matplotlib.pyplot as plt

# Assuming `pipeline` is your fitted model and it can predict probabilities

# Get the scores (probabilities) of the positive class

y\_scores = pipeline.predict\_proba(X\_test)[:, 1]

# Generate ROC curve values: false positive rates, true positive rates

fpr, tpr, thresholds = roc\_curve(y\_test, y\_scores)

# Calculate Area Under the Curve (AUC)

roc\_auc = auc(fpr, tpr)

# Plotting

plt.figure(figsize=(8, 6))

plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' % roc\_auc)

plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Receiver Operating Characteristic (ROC)')

plt.legend(loc="lower right")

plt.show()

**K fold cross validation**

from sklearn.model\_selection import cross\_val\_score, StratifiedKFold

# Initialize Stratified K-Fold to maintain the percentage of samples for each class

kfold = StratifiedKFold(n\_splits=5, shuffle=True, random\_state=42)

# Perform k-fold cross-validation

# Here we use 'accuracy' as the score to evaluate. You can choose other metrics like 'roc\_auc', 'f1', etc.

scores = cross\_val\_score(pipeline, X, y, cv=kfold, scoring='accuracy')

# Print the accuracy for each fold

print("Accuracy for each fold: ", scores)

# Print the mean accuracy across all folds

print("Mean cross-validation accuracy: ", scores.mean())

# Compare this mean accuracy to your baseline model's accuracy to check for improvement

from sklearn.dummy import DummyClassifier

# Assuming y is your target variable from the dataframe 'df'

dummy = DummyClassifier(strategy='most\_frequent', random\_state=42)

dummy\_scores = cross\_val\_score(dummy, X, y, cv=kfold, scoring='accuracy')

# Print the mean accuracy for the baseline model

print("Mean baseline accuracy: ", dummy\_scores.mean())

# Now you can compare it to your logistic regression model's accuracy

print("Mean logistic regression accuracy: ", scores.mean())

print("Improvement over baseline: ", scores.mean() - dummy\_scores.mean())

**NAÏVE BAYES:**

from sklearn.pipeline import make\_pipeline

from sklearn.preprocessing import StandardScaler

from sklearn.naive\_bayes import GaussianNB

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

from sklearn.metrics import confusion\_matrix, classification\_report

import seaborn as sns

import matplotlib.pyplot as plt

import pandas as pd

**Fitting the naive bayes model**

# Creating a pipeline that includes scaling and Gaussian Naive Bayes

pipeline = make\_pipeline(StandardScaler(), GaussianNB())

# Fitting the model to the training data

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

# Making predictions on the test set

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

# Evaluating the model

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

print(f"Test Accuracy: {accuracy}")

**confusion matrix**

from sklearn.metrics import confusion\_matrix, classification\_report

import seaborn as sns

import matplotlib.pyplot as plt

# Assuming y\_test are the true labels and y\_pred are the predictions made by the model

# Generating the confusion matrix

cm = confusion\_matrix(y\_test, y\_pred)

# Visualizing the confusion matrix

plt.figure(figsize=(8, 6))

sns.heatmap(cm, annot=True, fmt="d", cmap='Blues', xticklabels=['Predicted No', 'Predicted Yes'], yticklabels=['Actual No', 'Actual Yes'])

plt.ylabel('True label')

plt.xlabel('Predicted label')

plt.title('Confusion Matrix for Naive Bayes Model')

plt.show()

**recall, f1, precision**

from sklearn.metrics import classification\_report

# Generate and print the classification report for precision, recall, and F1 score

print(classification\_report(y\_test, y\_pred, target\_names=['No', 'Yes']))

# Calculate the confusion matrix to use for calculating classification error and specificity

cm = confusion\_matrix(y\_test, y\_pred)

tn, fp, fn, tp = cm.ravel()

# Classification error (also known as Misclassification Rate)

classification\_error = (fp + fn) / float(tp + tn + fp + fn)

print(f"Classification Error: {classification\_error:.2f}")

# Specificity (True Negative Rate)

specificity = tn / (tn + fp)

print(f"Specificity: {specificity:.2f}")

**ROC curve**

from sklearn.metrics import roc\_curve, roc\_auc\_score

import numpy as np

import matplotlib.pyplot as plt

# Step 1: Get the predicted probabilities for the positive class

y\_scores = pipeline.predict\_proba(X\_test)[:, 1]

# Step 2: Calculate ROC curve and AUC score

fpr, tpr, thresholds = roc\_curve(y\_test, y\_scores)

roc\_auc = roc\_auc\_score(y\_test, y\_scores)

# Step 3: Plot ROC curve

plt.figure(figsize=(8, 6))

plt.plot(fpr, tpr, label=f'ROC curve (AUC = {roc\_auc:.2f})')

plt.plot([0, 1], [0, 1], 'k--')  # Dashed diagonal

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('ROC Curve')

plt.legend(loc='lower right')

plt.show()

**adjusting the decision threshold based on Youden's J statistic**

from sklearn.metrics import roc\_curve

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

# Assume y\_scores are the predicted probabilities for the positive class

fpr, tpr, thresholds = roc\_curve(y\_test, y\_scores)

# Calculate Youden's J statistic

j\_statistic = tpr - fpr

optimal\_idx = np.argmax(j\_statistic)

optimal\_threshold = thresholds[optimal\_idx]

print(f"Optimal Threshold based on Youden's J statistic: {optimal\_threshold}")

# Step 4: Choose a new threshold from the ROC curve that suits your need

# This is a manual step depending on your specific requirement (e.g., balancing precision and recall)

# For demonstration, let's say you chose a threshold that gives you higher recall

new\_threshold = optimal\_threshold  # Example threshold

# Step 5: Apply the new threshold to adjust classification decisions

y\_pred\_adjusted = (y\_scores >= new\_threshold).astype(int)

# Now you can calculate metrics using the adjusted predictions to see the impact

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

print("Adjusted Metrics:")

print(f"Accuracy: {accuracy\_score(y\_test, y\_pred\_adjusted):.4f}")

print(f"Precision: {precision\_score(y\_test, y\_pred\_adjusted):.4f}")

print(f"Recall: {recall\_score(y\_test, y\_pred\_adjusted):.4f}")

print(f"F1 Score: {f1\_score(y\_test, y\_pred\_adjusted):.4f}")

**k-fold cross validation**

from sklearn.model\_selection import cross\_val\_score, StratifiedKFold

from sklearn.naive\_bayes import GaussianNB

from sklearn.pipeline import make\_pipeline

from sklearn.preprocessing import StandardScaler

# Assuming 'X' and 'y' are your features and target variable from the preprocessed DataFrame

# Creating a pipeline with a scaler and Gaussian Naive Bayes

pipeline = make\_pipeline(StandardScaler(), GaussianNB())

# Evaluating model performance with cross-validation

kfold = StratifiedKFold(n\_splits=5, shuffle=True, random\_state=42)

cv\_scores = cross\_val\_score(pipeline, X, y, cv=kfold, scoring='accuracy')

print(f"CV Accuracy: {cv\_scores.mean()} ± {cv\_scores.std()}")

**tuning hyperparameters**

from sklearn.model\_selection import GridSearchCV

from sklearn.naive\_bayes import GaussianNB

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import StandardScaler

# Assuming 'X' and 'y' are your features and target variable from the preprocessed DataFrame

# Creating a pipeline that includes scaling and Gaussian Naive Bayes

pipeline = Pipeline([

    ('scaler', StandardScaler()),

    ('naive\_bayes', GaussianNB())

])

# Define a grid of hyperparameters to search

# For GaussianNB, we can tune the 'var\_smoothing' parameter

param\_grid = {

    'naive\_bayes\_\_var\_smoothing': np.logspace(0, -9, num=100)

}

# Set up GridSearchCV

grid\_search = GridSearchCV(pipeline, param\_grid, cv=5, scoring='accuracy', verbose=1)

# Fit the GridSearchCV object to the data

grid\_search.fit(X, y)

# Print the best parameters and the best score

print("Best parameters found: ", grid\_search.best\_params\_)

print("Best cross-validation accuracy: ", grid\_search.best\_score\_)

**Clustering:**

**K-means clustering**

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

from sklearn.preprocessing import StandardScaler, LabelEncoder

from sklearn.cluster import KMeans

**Elbow method to find number of clusters**

# Assuming you've saved the cleaned data to 'cleaned\_data.csv'

data\_scaled = pd.read\_csv('/content/drive/MyDrive/dataset/Live\_scaled.csv')

# Select features for clustering

X = data\_scaled[['num\_reactions', 'num\_comments']].values

# Apply the Elbow Method to find the optimal number of clusters

wcss = []

for i in range(1, 11):

    kmeans = KMeans(n\_clusters=i, init='k-means++', n\_init=10, random\_state=42)

    kmeans.fit(X)

    wcss.append(kmeans.inertia\_)

plt.plot(range(1, 11), wcss)

plt.title('The Elbow Method')

plt.xlabel('Number of clusters')

plt.ylabel('WCSS')

plt.show()

**number of cluster fining using knee locator (optional)**

from kneed import KneeLocator

import matplotlib.pyplot as plt

# Number of clusters range

range\_n\_clusters = range(1, 11)

# Find the elbow point

knee\_locator = KneeLocator(range\_n\_clusters, wcss, curve='convex', direction='decreasing')

# Optimal number of clusters

optimal\_clusters = knee\_locator.elbow

print(f"The optimal number of clusters: {optimal\_clusters}")

**k-means cluster fitting**

# Perform K-means clustering

kmeans = KMeans(n\_clusters=3, init='k-means++', n\_init=10, random\_state=42)

y\_kmeans = kmeans.fit\_predict(X)

# Visualize the clusters

plt.scatter(X[y\_kmeans == 0, 0], X[y\_kmeans == 0, 1], s=100, c='red', label='Cluster 1')

plt.scatter(X[y\_kmeans == 1, 0], X[y\_kmeans == 1, 1], s=100, c='blue', label='Cluster 2')

plt.scatter(X[y\_kmeans == 2, 0], X[y\_kmeans == 2, 1], s=100, c='green', label='Cluster 3')

# Plotting the centroids of the clusters

plt.scatter(kmeans.cluster\_centers\_[:, 0], kmeans.cluster\_centers\_[:, 1], s=300, c='yellow', label='Centroids')

plt.title('K-means Clusters of Facebook Live Sellers')

plt.xlabel('Number of Reactions')

plt.ylabel('Number of Comments')

plt.legend()

plt.show()

**Print major element in each cluster:**

# Add cluster labels to the original cleaned data (not scaled)

data\_cleaned['Cluster'] = y\_kmeans

# Ensure this uses the original, unscaled 'data\_cleaned' with 'Cluster' labels assigned

for cluster in sorted(data\_cleaned['Cluster'].unique()):

    cluster\_data = data\_cleaned[data\_cleaned['Cluster'] == cluster]

    # Find the encoded majority status\_type, ensuring to work with non-scaled, encoded values

    majority\_status\_encoded = cluster\_data['status\_type'].mode()[0]

    # Decode the majority status\_type

    majority\_status = label\_encoder.inverse\_transform([int(majority\_status\_encoded)])[0]

    print(f"Majority status\_type in Cluster {cluster}: {majority\_status}")

**HIERARCHICAL CLUSTERING:**

**dendrogram to find the optimal number of clusters**

import scipy.cluster.hierarchy as sch

import matplotlib.pyplot as plt

# Create a dendrogram

dendrogram = sch.dendrogram(sch.linkage(X, method='ward'))

plt.title('Dendrogram')

plt.xlabel('Data Points')

plt.ylabel('Euclidean Distances')

plt.show()

**Hierarchical Clustering model fitting**

from sklearn.cluster import AgglomerativeClustering

# Perform hierarchical clustering

hc = AgglomerativeClustering(n\_clusters=3, affinity='euclidean', linkage='ward')

y\_hc = hc.fit\_predict(X)

**Visualizing the clusters**

# Visualize the clusters

plt.scatter(X[y\_hc == 0, 0], X[y\_hc == 0, 1], s=100, c='red', label='Cluster 1')

plt.scatter(X[y\_hc == 1, 0], X[y\_hc == 1, 1], s=100, c='blue', label='Cluster 2')

plt.scatter(X[y\_hc == 2, 0], X[y\_hc == 2, 1], s=100, c='green', label='Cluster 3')

plt.title('Clusters of Facebook Live Sellers (Hierarchical Clustering)')

plt.xlabel('Number of Reactions')

plt.ylabel('Number of Comments')

plt.legend()

plt.show()

**Majority status type in each cluster**

# Add the cluster labels to your original dataset

data['Cluster'] = y\_hc

# Label encode 'status\_type' to revert back to original categorical types for majority voting

label\_encoder = LabelEncoder()

data['status\_type\_encoded'] = label\_encoder.fit\_transform(data['status\_type'])

# Calculate the majority 'status\_type' for each cluster

for i in range(3):  # assuming 3 clusters

    cluster\_group = data[data['Cluster'] == i]

    majority\_status = cluster\_group['status\_type'].value\_counts().idxmax()

    print(f"Majority status\_type in Cluster {i}: {majority\_status}")

**Decision Tree:**

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score

dt = DecisionTreeClassifier(random\_state=42)

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

y\_pred\_dt = dt.predict(X\_test)

print(f"Decision Tree Accuracy: {accuracy\_score(y\_test, y\_pred\_dt)}")

**hyperparameter tuning**

from sklearn.model\_selection import GridSearchCV

dt\_param\_grid = {

    'max\_depth': [None, 10, 20, 30],

    'min\_samples\_leaf': [1, 2, 4],

    'min\_samples\_split': [2, 5, 10]

}

dt\_grid\_search = GridSearchCV(DecisionTreeClassifier(random\_state=42), dt\_param\_grid, cv=5, scoring='accuracy')

dt\_grid\_search.fit(X\_train, y\_train)

print(f"Best parameters for Decision Tree: {dt\_grid\_search.best\_params\_}")

# Using the best estimator from the grid search

best\_dt = dt\_grid\_search.best\_estimator\_

y\_pred\_best\_dt = best\_dt.predict(X\_test)

print(f"Decision Tree Accuracy after tuning: {accuracy\_score(y\_test, y\_pred\_best\_dt)}")

**Neural networks:**

from sklearn.neural\_network import MLPClassifier

nn = MLPClassifier(random\_state=42, max\_iter=1000)

nn.fit(X\_train, y\_train)  # Ensure X\_train is scaled

y\_pred\_nn = nn.predict(X\_test)

print(f"Neural Network Accuracy: {accuracy\_score(y\_test, y\_pred\_nn)}")

**hyperparameter tuning**

nn\_param\_grid = {

    'hidden\_layer\_sizes': [(50,), (100,), (50,50), (100,100)],

    'activation': ['tanh', 'relu'],

    'solver': ['sgd', 'adam'],

    'alpha': [0.0001, 0.05],

}

nn\_grid\_search = GridSearchCV(MLPClassifier(random\_state=42, max\_iter=1000), nn\_param\_grid, cv=5, scoring='accuracy')

nn\_grid\_search.fit(X\_train, y\_train)

print(f"Best parameters for Neural Network: {nn\_grid\_search.best\_params\_}")

best\_nn = nn\_grid\_search.best\_estimator\_

y\_pred\_best\_nn = best\_nn.predict(X\_test)

print(f"Neural Network Accuracy after tuning: {accuracy\_score(y\_test, y\_pred\_best\_nn)}")