Machine learning

📦 Section 1: Importing Libraries and Dependencies

1.1 Making the split

```
from sklearn.model_selection import train_test_split
X = df.drop('target', axis=1)
y = df['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

If you want to call more than one in a df you put double square brackets instead of single

Before starting, make sure you have installed all the necessary packages:

1.1 Required Package Installations

```
pip install pandas numpy scikit-learn matplotlib seaborn plotly
```

1.2 Importing Libraries

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
from sklearn.model_selection import train_test_split, cross_val_score,
GridSearchCV
from sklearn.preprocessing import StandardScaler, MinMaxScaler, LabelEncoder,
OneHotEncoder
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report, f1_score, roc_auc_score, roc_curve,
precision_recall_curve, auc
from sklearn.metrics import precision_score, recall_score,
```

```
ConfusionMatrixDisplay
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
```

🛠 Section 2: Data Loading and Initial Exploration

2.1 Loading the Data

```
# Load dataset
df = pd.read_csv('your_dataset.csv')

# Display the first few rows
print(df.head())
```

2.2 Understanding the Data

```
# Checking the shape of the dataset
print(f"Dataset Shape: {df.shape}")

# Checking for missing values
print(df.isnull().sum())

# Getting summary statistics
print(df.describe())

# Checking data types of each column
print(df.info())
```

Explanation:

- `.shape` gives the number of rows and columns.
- `.isnull().sum()` identifies columns with missing values.
- `.describe()` provides summary statistics (mean, median, etc.).
- `.info()` shows data types and non-null counts.

🔍 Section 3: Data Cleaning & Preprocessing

3.1 Handling Missing Values

```
# Fill missing numerical values with the median
df.fillna(df.median(), inplace=True)

# Fill missing categorical values with the mode
df.fillna(df.mode().iloc[0], inplace=True)
```

Explanation:

- **Median** is used for numerical data to avoid skewing due to outliers.
- **Mode** is used for categorical features to fill with the most frequent value.

3.2 Removing Duplicates

```
# Remove duplicate rows
df.drop_duplicates(inplace=True)
print(f"Data after removing duplicates: {df.shape}")
```

🧹 Section 4: Feature Encoding

4.1 Label Encoding (For Binary/Ordinal Categories)

```
# Initialize LabelEncoder
le = LabelEncoder()

# Apply label encoding to binary columns
df['binary_column'] = le.fit_transform(df['binary_column'])
```

Explanation: Label encoding converts categorical values into integers (0, 1, 2, etc.), useful for binary classification.

4.2 One-Hot Encoding (For Nominal Categories)

```
# Apply one-hot encoding
df = pd.get_dummies(df, columns=['nominal_column'], drop_first=True)
```

Explanation: One-hot encoding converts categorical values into multiple binary columns (dummy variables), avoiding ordinal relationships.

🔄 Section 5: Feature Scaling

5.1 Standardization

```
scaler = StandardScaler()
numerical_features = ['feature1', 'feature2', 'feature3']
df[numerical_features] = scaler.fit_transform(df[numerical_features])
```

Explanation:

- Standardization transforms data to have a **mean of 0** and **standard deviation of 1**.
- Useful for models like **SVM, KNN, and Neural Networks**.

5.2 Normalization (Min-Max Scaling)

```
min_max_scaler = MinMaxScaler()
df[numerical_features] = min_max_scaler.fit_transform(df[numerical_features])
```

Explanation:

- Normalization scales features between **0 and 1**.
- Useful for algorithms like **Neural Networks** that are sensitive to feature scales.

5.3 Log Transformation

```
df['skewed_feature'] = np.log1p(df['skewed_feature'])
```

Explanation: Log transformation is used to **reduce skewness** in data.

📊 Section 6: Data Visualization for EDA

6.1 Distribution Plot

```
# Plotting the distribution of a feature
sns.histplot(df['numerical_feature'], kde=True)
plt.title('Distribution of Numerical Feature')
plt.show()
```

6.2 Correlation Heatmap

```
plt.figure(figsize=(10, 8))
sns.heatmap(df.corr(), annot=True, cmap='coolwarm')
plt.title('Correlation Matrix')
plt.show()
```

6.3 Boxplot for Outliers

```
sns.boxplot(x=df['numerical_feature'])
plt.title('Boxplot for Outliers')
plt.show()
```

6.4 Pairplot for Feature Relationships

```
sns.pairplot(df, hue='target')
plt.title('Pairplot')
plt.show()
```

🔍 Section 7: Splitting Data into Training and Testing Sets

```
X = df.drop('target', axis=1)
y = df['target']

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
stratify=y, random_state=42)
```

🤖 Section 8: Training Machine Learning Models

8.1 Logistic Regression

```
model = LogisticRegression()
model.fit(X_train, y_train)
log_preds = model.predict(X_test)

print(f"Logistic Regression Accuracy: {accuracy_score(y_test,
log_preds):.2f}")
```

8.2 Random Forest

```
rf_model = RandomForestClassifier(n_estimators=100)
rf_model.fit(X_train, y_train)
rf_preds = rf_model.predict(X_test)
```

8.3 Support Vector Machine (SVM)

```
svm_model = SVC(probability=True)
svm_model.fit(X_train, y_train)
svm_preds = svm_model.predict(X_test)
```

✅ Section 9: Model Evaluation

9.1 Confusion Matrix

```
cm = confusion_matrix(y_test, rf_preds)
ConfusionMatrixDisplay(confusion_matrix=cm).plot()
plt.title('Confusion Matrix')
plt.show()
```

9.2 Classification Report

```
print(classification_report(y_test, rf_preds))
```

9.3 ROC Curve and AUC

```
y_proba = rf_model.predict_proba(X_test)[:, 1]
fpr, tpr, _ = roc_curve(y_test, y_proba)
roc_auc = auc(fpr, tpr)

plt.plot(fpr, tpr, label=f'AUC = {roc_auc:.2f}')
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.show()
```

9.4 Precision-Recall Curve

```
precision, recall, _ = precision_recall_curve(y_test, y_proba)
plt.plot(recall, precision)
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.show()
```

10.1 Cross-Validation

```
cv_scores = cross_val_score(rf_model, X, y, cv=5)
print(f"Cross-Validation Accuracy: {np.mean(cv_scores):.2f}")
```

10.2 Hyperparameter Tuning with GridSearchCV

```
param_grid = {'n_estimators': [50, 100, 200], 'max_depth': [None, 10, 20]}
grid_search = GridSearchCV(RandomForestClassifier(), param_grid, cv=5)
grid_search.fit(X_train, y_train)
print(f"Best Params: {grid_search.best_params_}")
```

---

1. Decision Trees

What is a Decision Tree?

- A Decision Tree is a supervised machine learning algorithm used for both classification and regression tasks.
- It splits the dataset into subsets based on the most significant feature at each step, forming a tree-like structure of decisions.
- The nodes represent features, the branches represent decision rules, and the leaves represent outcomes (class labels or values).

How Does a Decision Tree Work?

1. **Root Node**: The initial node, which represents the entire dataset, is split using the best feature.
2. **Decision Nodes**: Nodes where the dataset is split further.
3. **Leaf Nodes**: Terminal nodes that represent the final output (class or value).

4. **Splitting Criteria**: Measures like **Gini Impurity**, **Entropy (Information Gain)**, and **Mean Squared Error (for regression)** are used to determine the best splits.

Implementation of Decision Tree Classifier

Code Example

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.metrics import accuracy_score, classification_report,
ConfusionMatrixDisplay
import matplotlib.pyplot as plt
import seaborn as sns

# Load the dataset (example using the UGRansome dataset)
df = pd.read_csv('ugransome.csv')
df['sentiment_encoded'] = df['sentiment'].astype('category').cat.codes

# Define features and target variable
X = df[['score', 'sentiment_encoded']]
y = df['nature'].astype('category').cat.codes

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Train a Decision Tree model
dt_model = DecisionTreeClassifier(max_depth=5, random_state=42)
dt_model.fit(X_train, y_train)

# Predict on the test set
y_pred = dt_model.predict(X_test)

# Evaluate the model
print("Accuracy:", accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))
ConfusionMatrixDisplay.from_estimator(dt_model, X_test, y_test)
plt.show()
```

How to Plot a Decision Tree

Visualizing the Decision Tree

```
plt.figure(figsize=(15, 10))
plot_tree(dt_model, feature_names=X.columns, class_names=['Class 0', 'Class
1'], filled=True)
plt.title("Decision Tree Visualization")
plt.show()
```

Explanation:

- The `plot_tree` function visualizes the splits, showing the feature, threshold, Gini/Entropy value, and the samples at each node.
- **Feature importance** can also be plotted to show which features have the most influence on the model.

```
# Plot feature importance
importances = pd.Series(dt_model.feature_importances_, index=X.columns)
importances.sort_values(ascending=True).plot(kind='barh')
plt.title("Feature Importance")
plt.show()
```

---

2. Logistic Regression

What is Logistic Regression?

- Logistic Regression is a supervised learning algorithm used for binary classification.
- It models the probability that an instance belongs to a particular class using the **logistic function (sigmoid)**.
- It outputs a probability between 0 and 1, which can be thresholded to classify data points.

Sigmoid Function

[
\text{Sigmoid}(z) = \frac{1}{1 + e^{-z}}
]

- **z** is the linear combination of input features (i.e., (z = w_1x_1 + w_2x_2 + ... + b)).

Implementation of Logistic Regression

Code Example

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_curve, auc, confusion_matrix

# Train a Logistic Regression model
log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)

# Make predictions
y_pred = log_reg.predict(X_test)
y_pred_proba = log_reg.predict_proba(X_test)[:, 1]

# Evaluate the model
print("Accuracy:", accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))
ConfusionMatrixDisplay.from_estimator(log_reg, X_test, y_test)
plt.show()

# Plot ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)
roc_auc = auc(fpr, tpr)

plt.figure()
plt.plot(fpr, tpr, label=f'ROC curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.show()
```

Explanation:

- The **ROC curve** shows the trade-off between sensitivity (True Positive Rate) and specificity (1 - False Positive Rate).
- The **AUC (Area Under Curve)** provides a single value summary of the model's performance.

---

3. Support Vector Machines (SVM)

What is SVM?

- SVM is a supervised learning algorithm used for classification and regression.
- It finds the hyperplane that best separates the data into classes, maximizing the margin between classes.
- It is effective in high-dimensional spaces and is robust to outliers using the **soft margin**.

Kernel Trick

- **Linear Kernel**: Used when data is linearly separable.
- **Polynomial/RBF Kernel**: Used when data is not linearly separable.

Plotting SVM Decision Boundaries

Code Example

```
from sklearn.svm import SVC
from sklearn.metrics import classification_report
import numpy as np

# Train an SVM model with RBF kernel
svm_model = SVC(kernel='rbf', C=1.0, probability=True)
svm_model.fit(X_train, y_train)

# Predict on test data
y_pred = svm_model.predict(X_test)
print(classification_report(y_test, y_pred))

# Plot SVM decision boundary (for 2D data only)
def plot_svm_boundary(X, y, model):
    plt.figure(figsize=(10, 6))
    h = 0.02  # Step size in the mesh
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max,
h))

    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.contourf(xx, yy, Z, alpha=0.8)
    sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=y, edgecolor='k')
```

```
    plt.title("SVM Decision Boundary")
    plt.show()

# Plot the decision boundary
X_train_2d = X_train[:, :2]  # Use only 2 features for visualization
plot_svm_boundary(X_train_2d, y_train, svm_model)
```

Explanation:

- The `plot_svm_boundary` function visualizes how the SVM model separates the classes using a decision boundary.
- The **support vectors** (data points closest to the hyperplane) are critical for defining the boundary.

Summary of Key Differences

Model	Use Case	Advantages	Limitations
Decision Tree	Classification & Regression	Easy to interpret, handles non-linearity	Prone to overfitting
Logistic Regression	Binary Classification	Simple, interpretable, probabilistic output	Assumes linearity
SVM	Classification & Regression	Effective in high dimensions, works well with outliers	Expensive to train with large data

---

Regresion Models
When working with **continuous data**, you're typically dealing with **regression problems**, where the target variable is a continuous numerical value (e.g., predicting house prices, temperature, stock prices, etc.).

Below is a list of models that work well with continuous data:

---

1. Linear Models For Continuous data

1.1 Linear Regression

- **Description**: Fits a straight line to minimize the difference between actual and predicted values.
- **Use Case**: When there's a linear relationship between features and the target.
- **Implementation**:

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

1.2 Ridge Regression (L2 Regularization)

- **Description**: Regularized version of Linear Regression that penalizes large coefficients.
- **Use Case**: Reduces overfitting in linear models.
- **Implementation**:

```
from sklearn.linear_model import Ridge
model = Ridge(alpha=1.0)
model.fit(X_train, y_train)
```

1.3 Lasso Regression (L1 Regularization)

- **Description**: Similar to Ridge but can shrink some coefficients to zero, performing feature selection.
- **Use Case**: When you want to select important features automatically.
- **Implementation**:

```
from sklearn.linear_model import Lasso
model = Lasso(alpha=0.1)
model.fit(X_train, y_train)
```

1.4 Polynomial Regression

- **Description**: Extends Linear Regression by adding polynomial features.
- **Use Case**: When the relationship between features and target is non-linear.
- **Implementation**:

```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=3)
X_poly = poly.fit_transform(X)
model = LinearRegression().fit(X_poly, y)
```

---

2. Tree-Based Models

2.1 Decision Tree Regression

- **Description**: Splits data into branches to predict continuous values.
- **Use Case**: Works well for capturing non-linear relationships.
- **Implementation**:

```
from sklearn.tree import DecisionTreeRegressor
model = DecisionTreeRegressor(max_depth=5)
model.fit(X_train, y_train)
```

2.2 Random Forest Regression

- **Description**: Ensemble model using multiple decision trees for better generalization.
- **Use Case**: Reduces overfitting and handles complex data.
- **Implementation**:

```
from sklearn.ensemble import RandomForestRegressor
model = RandomForestRegressor(n_estimators=100)
model.fit(X_train, y_train)
```

2.3 Gradient Boosting Regression

- **Description**: Builds an ensemble of trees in a sequential manner, reducing errors of the previous trees.
- **Use Case**: Provides high accuracy, often used in competitions.
- **Implementation**:

```
from sklearn.ensemble import GradientBoostingRegressor
model = GradientBoostingRegressor()
```

```
model.fit(X_train, y_train)
```

2.4 XGBoost Regression

- **Description**: An optimized version of Gradient Boosting that is faster and more efficient.
- **Use Case**: When you need high accuracy and efficiency.
- **Implementation**:

```
from xgboost import XGBRegressor
model = XGBRegressor()
model.fit(X_train, y_train)
```

---

3. Support Vector Machines

3.1 Support Vector Regression (SVR)

- **Description**: Uses kernel functions to fit the data within a margin.
- **Use Case**: Effective in high-dimensional spaces and for non-linear data.
- **Implementation**:

```
from sklearn.svm import SVR
model = SVR(kernel='rbf')
model.fit(X_train, y_train)
```

---

4. Neural Networks

4.1 Multi-Layer Perceptron (MLP) Regression

- **Description**: A type of neural network that can learn complex patterns.
- **Use Case**: Useful for modeling non-linear relationships and high-dimensional data.
- **Implementation**:

```
from sklearn.neural_network import MLPRegressor
model = MLPRegressor(hidden_layer_sizes=(50, 30, 10))
```

```
model.fit(X_train, y_train)
```

4.2 Deep Learning Models (Keras/TensorFlow)

- **Description**: Custom neural networks using deep learning frameworks.
- **Use Case**: Useful for large datasets and complex non-linear relationships.
- **Implementation**:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(X_train.shape[1],)))
model.add(Dense(32, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
model.fit(X_train, y_train, epochs=20)
```

---

5. K-Nearest Neighbors Regression

5.1 KNN Regression

- **Description**: Predicts values based on the average of the k-nearest neighbors.
- **Use Case**: Simple and interpretable, but sensitive to noisy data.
- **Implementation**:

```
from sklearn.neighbors import KNeighborsRegressor
model = KNeighborsRegressor(n_neighbors=5)
model.fit(X_train, y_train)
```

---

6. Evaluation Metrics for Regression

6.1 R^2 Score (Coefficient of Determination)

- Measures how well the model explains the variance of the target variable.
- **Implementation**:

```
from sklearn.metrics import r2_score
print(f"R² Score: {r2_score(y_test, y_pred)}")
```

6.2 Mean Squared Error (MSE)

- Measures the average squared difference between actual and predicted values.
- **Implementation**:

```
from sklearn.metrics import mean_squared_error
print(f"MSE: {mean_squared_error(y_test, y_pred)}")
```

6.3 Mean Absolute Error (MAE)

- Measures the average absolute difference between actual and predicted values.
- **Implementation**:

```
from sklearn.metrics import mean_absolute_error
print(f"MAE: {mean_absolute_error(y_test, y_pred)}")
```

6.4 Root Mean Squared Error (RMSE)

- Provides a metric in the same units as the target variable.
- **Implementation**:

```
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print(f"RMSE: {rmse}")
```

Summary

Model	Best For
Linear Regression	Simple linear relationships

Model	Best For
Ridge/Lasso Regression	Reducing overfitting
Polynomial Regression	Non-linear relationships
Decision Tree Regression	Non-linear, interpretable models
Random Forest	Robust, reduces overfitting
Gradient Boosting/XGBoost	High accuracy, competition-level models
SVR	High-dimensional, non-linear data
Neural Networks (MLP)	Complex patterns, non-linear data
KNN Regression	Simple, interpretable