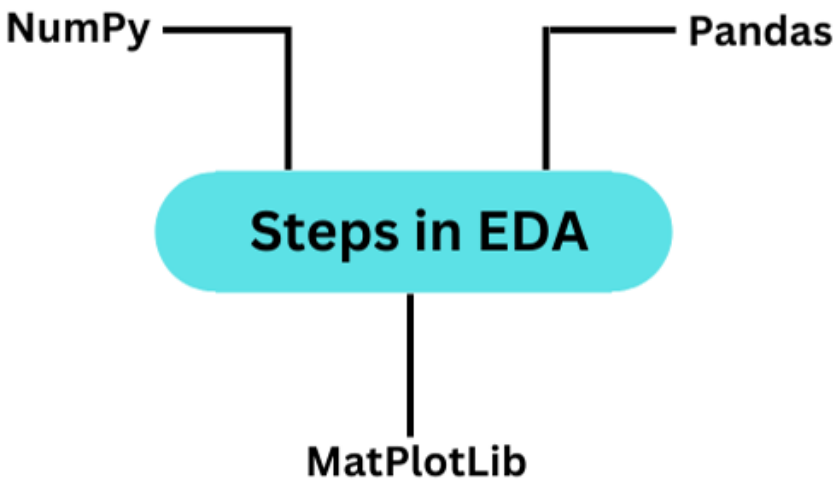
# EDA - Exploratory Data Analysis

## Why do we do EDA???

- Examine the data distribution
- Handling missing values of the dataset(a most common issue with every dataset)
- Handling the outliers
- Removing duplicate data
- Encoding the categorical variables
- Normalizing and Scaling




## NUMPY

NumPy arrays are faster and more compact than Python lists. An array consumes less memory and is convenient to use.

## How to import NumPy

```
import numpy as np
```

## How to create a basic array

1.

```
import numpy as np
a = np.array([1, 2, 3])
```


2.

```
>>> np.zeros(2)
array([0., 0.])
```

3.

```
>>> np.ones(2)
array([1., 1.])
```

4. empty()

The function `empty` creates an array whose initial content is random and depends on the state of the memory.

```
>>> # Create an empty array with 2 elements
>>> np.empty(2)
array([3.14, 42.  ])  # may vary
```

#Arrange the elements

```
>>> np.arange(2, 9, 2)
array([2, 4, 6, 8])
```

## #Linspace

```
>>> np.linspace(0, 10, num=5)
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

## #Array Attributes

a = np.array([1, 2, 3])

```
a.shape  # (3,)
a.size   # 3
a.ndim   # 1
a.dtype  # dtype('int64')
a.itemsize  # 8
a.nbytes  # 24
```

## # Array Manipulation

```
b.reshape((3, 2))
b.ravel()
b.flatten()
```

### Concatenation and Splitting:

```
np.concatenate((a, a))
np.hstack((a, a))
np.vstack((a, a))
np.split(a, 3)
```

### 4. Indexing and Slicing:

- **Basic Slicing: Use : to slice arrays**

```
a[1:3]  # [2, 3]
b[:, 1]  # All rows, column 1
```

- **Advanced Indexing:** Boolean indexing, integer array indexing.

```
a[a > 1]  # [2, 3]
b[[0, 1], [1, 2]]  # [0., 0.]
```

### 5 . Mathematical Operations:

- **Element-wise Operations:** +, -, *, /, **.

```
a + 2  # [3, 4, 5]
a * 2  # [2, 4, 6]
```

- **Aggregate Functions:** `np.sum()`, `np.mean()`, `np.std()`, `np.var()`, `np.min()`, `np.max()`, `np.argmin()`, `np.argmax()`.

```
np.sum(a)  # 6
np.mean(a)  # 2.0
np.median(a)  # 2.5
np.std(a)  # 1.0
```

- **Linear Algebra:** `np.dot()`, `np.matmul()`, `np.linalg.inv()`, `np.linalg.eig()`, `np.linalg.svd()`.

```
np.dot(a, a)
np.linalg.inv(np.array([[1, 2], [3, 4]]))
np.linalg.det(np.array([[1, 2], [3, 4]]))
```

**6. Random Numbers:**

- **Random Number Generation:** `np.random.seed()`, `np.random.rand()`, `np.random.randn()`, `np.random.randint()`, `np.random.choice()`, `np.random.permutation()`.

```
np.random.seed(42)
np.random.rand(3, 2)
np.random.randn(3, 2)
np.random.randint(0, 10, (3, 3))
np.random.choice([1, 2, 3, 4], 3)
```

## Application in Machine Learning:

**1. Data Preparation:**

- **Loading Data:** Use NumPy functions to load data from files (`np.loadtxt()`, `np.genfromtxt()`, `np.load()`, `np.save()`).

**Normalization/Standardization:** Normalize features for better performance of ML algorithms.

```
X = (X - np.mean(X, axis=0)) / np.std(X, axis=0)
```

**2. Model Implementation:**

**Vectorized Operations:** Implement ML algorithms efficiently using NumPy for operations like matrix multiplication and broadcasting.

```
weights = np.random.rand(n_features)
predictions = np.dot(X, weights)
errors = predictions - y
mse = np.mean(errors ** 2)
mse
```

**3. Performance Evaluation:**

**Metrics Calculation:** Use NumPy to calculate performance metrics like accuracy, precision, recall, etc.

```
accuracy = np.sum(y_pred == y_true) / len(y_true)
accuracy
```

**4. Optimization:**

**Gradient Descent:** Efficiently compute gradients and update parameters using NumPy.

```
gradients = np.dot(X.T, (predictions - y)) / m
weights -= learning_rate * gradients
```

# PANDAS

Pandas is a powerful library for data manipulation and analysis in Python. It is widely used in Machine Learning (ML) for preprocessing data, exploring datasets, and performing complex operations with ease. Here are detailed notes on using Pandas for ML:

**Overview of Pandas:**

- **Data Structures:** Main data structures are `Series` (1-dimensional) and `DataFrame` (2-dimensional).
- **Integration:** Works seamlessly with NumPy and other data science libraries like SciPy, scikit-learn, and Matplotlib.
- **Functionality:** Provides tools for reading/writing data, cleaning, merging, reshaping, and aggregating.

## Key Features and Functions:

## 1. Data Structures:

### Series:

**Creating Series:**

```
import pandas as pd

s = pd.Series([1, 2, 3, 4])

s = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])
```

## DataFrame:

**Creating DataFrames:**

```
data = {'A': [1, 2, 3], 'B': [4, 5, 6]}

df = pd.DataFrame(data)

df = pd.DataFrame(data, index=['row1', 'row2', 'row3'])
```

**2. Data Inspection:**

**Basic Information:**

```
df.head()  # First 5 rows

df.tail()  # Last 5 rows

df.info()  # Data types, non-null counts

df.describe()  # Summary statistics

df.shape  # Dimensions of the DataFrame

df.columns  # Column names

df.index  # Index names
```

**3. Data Selection and Indexing:**

**Selecting Data:**

```
df['A']  # Select a single column

df[['A', 'B']]  # Select multiple columns

df.loc['row1']  # Select a row by label

df.iloc[0]  # Select a row by position

df.loc['row1', 'A']  # Select a single value by label

df.iloc[0, 0]  # Select a single value by position
```

**4. Data Cleaning:**

**Handling Missing Values:**

```
df.isnull().sum()  # Count missing values

df.dropna()  # Remove rows with missing values

df.fillna(value)  # Fill missing values
```

```
df['A'].fillna(df['A'].mean(), inplace=True)  # Fill missing values with mean
```

**Removing Duplicates:**

```
df.drop_duplicates(inplace=True)
```

#### 5. Data Transformation:

**Renaming Columns:**

```
df.rename(columns={'A': 'a'}, inplace=True)
```

**Changing Data Types:**

```
df['A'] = df['A'].astype(float)
```

**Applying Functions:**

```
df['A'] = df['A'].apply(lambda x: x + 1)

df['C'] = df.apply(lambda row: row['A'] + row['B'], axis=1)
```

#### 6. Data Aggregation and Grouping:

**Grouping Data:**

```
grouped = df.groupby('A')

grouped.mean()

grouped.agg(['mean', 'sum'])
```

**Pivot Tables:**

```
pivot_table = df.pivot_table(values='B', index='A', columns='C', aggfunc='mean')
```

**7. Data Merging and Joining:**

**Merging DataFrames:**

```
df1 = pd.DataFrame({'key': ['A', 'B'], 'value': [1, 2]})

df2 = pd.DataFrame({'key': ['B', 'C'], 'value': [3, 4]})

merged = pd.merge(df1, df2, on='key', how='inner')
```

**Joining DataFrames:**

```
df1.set_index('key', inplace=True)

df2.set_index('key', inplace=True)

joined = df1.join(df2, how='inner')
```

**8. Time Series Data:**

**Date Range and Frequency:**

```
date_rng = pd.date_range(start='2020-01-01', end='2020-01-10',
freq='D')

df = pd.DataFrame(date_rng, columns=['date'])

df['data'] = np.random.randint(0, 100, size=(len(date_rng)))
```

**Datetime Operations:**

```
df['date'] = pd.to_datetime(df['date'])

df.set_index('date', inplace=True)

df['month'] = df.index.month

df['day_of_week'] = df.index.dayofweek
```

## Application in Machine Learning:

### 1. Data Preparation:

#### Loading Data:

```
df = pd.read_csv('data.csv')

df = pd.read_excel('data.xlsx')

df = pd.read_sql('SELECT * FROM table', connection)
```

#### Feature Engineering:

```
df['new_feature'] = df['A'] * df['B']

df['category_encoded'] =
df['category'].astype('category').cat.codes
```

### 2. Exploratory Data Analysis (EDA):

#### Descriptive Statistics:

```
df.describe()

df['A'].value_counts()

df.corr()
```

#### Data Visualization:

```
import matplotlib.pyplot as plt

df['A'].hist()

df.plot(x='A', y='B', kind='scatter')
```

### 3. Data Preprocessing:

**Normalization/Standardization:**

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

df[['A', 'B']] = scaler.fit_transform(df[['A', 'B']])
```

**Handling Categorical Variables:**

```
df = pd.get_dummies(df, columns=['category'])
```

**4. Model Implementation:**

**Splitting Data:**

```
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)
```

**Training a Model:**

```
from sklearn.linear_model import LinearRegression

model = LinearRegression()

model.fit(X_train, y_train)

predictions = model.predict(X_test)
```

**5. Performance Evaluation:**

**Calculating Metrics:**

```
from sklearn.metrics import mean_squared_error, r2_score

mse = mean_squared_error(y_test, predictions)
```

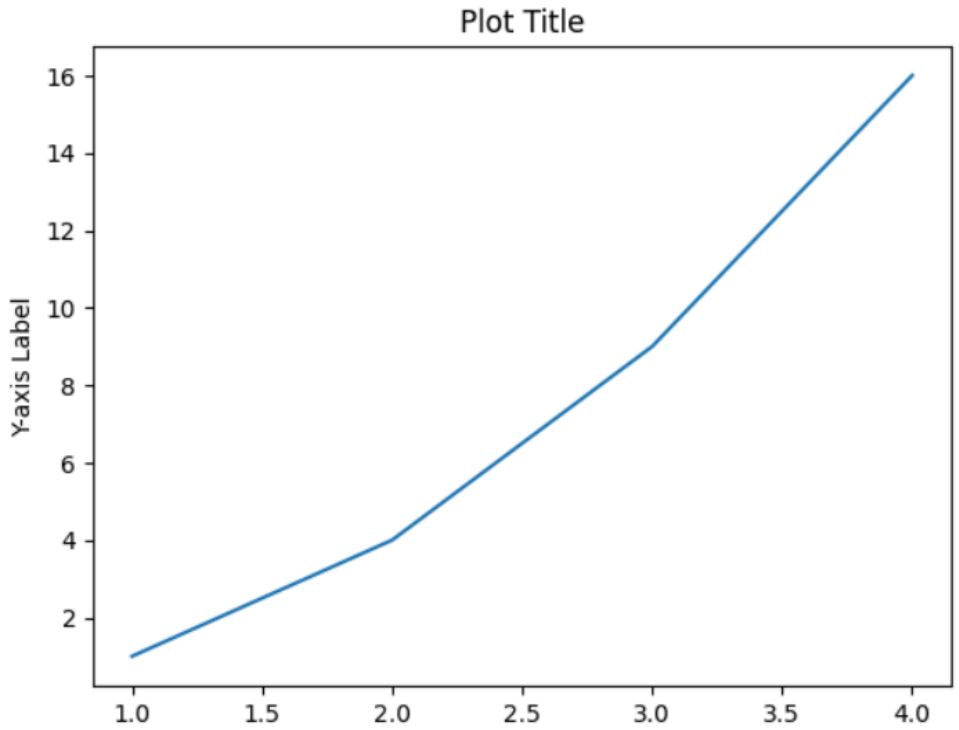
- r2 = r2_score(y_test, predictions)

## MATPLOTLIB

Matplotlib is a widely-used plotting library in Python for creating static, interactive, and animated visualizations. It is an essential tool in the data analysis and machine learning workflow for visualizing data and understanding relationships within the data. Here are detailed notes on using Matplotlib for machine learning:

1. **Basic Plotting:**

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
plt.xlabel('X-axis Label')
plt.ylabel('Y-axis Label')
plt.title('Plot Title')
plt.show()
```




**2. Common Plots:**

**Line Plot:**

```
plt.plot(x, y, label='Line Label')

plt.legend()
```

```
plt.show()
```

**Scatter Plot:**

python

Copy code

```
plt.scatter(x, y)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Scatter Plot')
plt.show()
```

**Bar Plot:**

python

Copy code

```
plt.bar(categories, values)
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Bar Plot')
plt.show()
```

**Histogram:**

```
plt.hist(data, bins=10)
plt.xlabel('Bins')
plt.ylabel('Frequency')
```

```
plt.title('Histogram')
plt.show()
```

**Box Plot:**

```
plt.boxplot(data)
plt.xlabel('X-axis Label')
plt.ylabel('Y-axis Label')
plt.title('Box Plot')
plt.show()
```

**Heatmap:**

```
import numpy as np
import seaborn as sns

data = np.random.rand(10, 10)
sns.heatmap(data, annot=True)
plt.title('Heatmap')
plt.show()
```

**3. Customization:**

**Plot Style:**

```
plt.style.use('ggplot')  # Use a predefined style
```

**Labels and Title:**

```
plt.xlabel('X-axis Label')

plt.ylabel('Y-axis Label')

plt.title('Plot Title')
```

**Legend:**

```
plt.plot(x, y, label='Line Label')

plt.legend(loc='upper left')
```

**Annotations:**

```
plt.annotate('Annotation Text', xy=(x_point, y_point),
xytext=(x_text, y_text),

             arrowprops=dict(facecolor='black',
arrowstyle='->'))
```

**Grid:**

```
plt.grid(True)
```

**Colors and Markers:**

```
plt.plot(x, y, color='r', marker='o', linestyle='--')
```

#### Axis Limits:

```
plt.xlim(0, 10)

plt.ylim(0, 20)
```

## Subplots:

### Creating Multiple Subplots:

```
fig, axs = plt.subplots(2, 2)

axs[0, 0].plot(x, y)

axs[0, 1].scatter(x, y)

axs[1, 0].bar(categories, values)

axs[1, 1].hist(data)

plt.tight_layout()  # Adjust subplots to fit into the figure
area.

plt.show()
```

#### 4. Saving Plots:

```
plt.savefig('plot.png')  # Save the plot as a PNG file

plt.savefig('plot.pdf')  # Save the plot as a PDF file
```

## Application in Machine Learning:

#### 1. Exploratory Data Analysis (EDA):

#### Visualizing Data Distributions:

### Histograms and Box Plots:

```
plt.hist(data['feature'], bins=30)
plt.title('Feature Distribution')
plt.show()

plt.boxplot(data['feature'])
plt.title('Feature Box Plot')
plt.show()
```

#### Correlation Analysis:

### Heatmap of Correlation Matrix:

```
corr_matrix = data.corr()
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm')
plt.title('Correlation Matrix Heatmap')
plt.show()
```

#### 2. Model Evaluation:

#### Plotting Model Performance:

### Training vs. Validation Loss:

```
plt.plot(epochs, train_loss, label='Training Loss')
plt.plot(epochs, val_loss, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
```

```
plt.title('Training and Validation Loss')

plt.legend()

plt.show()
```

**Confusion Matrix:**

```
from sklearn.metrics import confusion_matrix

import seaborn as sns

cm = confusion_matrix(y_test, y_pred)

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')

plt.xlabel('Predicted Label')

plt.ylabel('True Label')

plt.title('Confusion Matrix')

plt.show()
```

**3. Feature Importance:**

**Bar Plot of Feature Importance:**

```
importance = model.feature_importances_

indices = np.argsort(importance)[::-1]

features = data.columns


plt.bar(range(len(indices)), importance[indices])

plt.xticks(range(len(indices)), [features[i] for i in
indices], rotation=90)
```

```
plt.title('Feature Importance')

plt.show()
```

## K-Nearest Neighbors (KNN) Algorithm

K-Nearest Neighbors (KNN) is a simple, non-parametric, and lazy learning algorithm used for classification and regression tasks in machine learning. It operates on the principle that similar instances exist in close proximity to each other.

**Key Concepts:**

1. **Instance-Based Learning:** KNN is an instance-based learning algorithm where the model is essentially the entire training dataset. Predictions for new data points are made by comparing them to the training instances.
2. **Distance Metrics:** KNN relies on distance metrics to find the "nearest neighbors" of a given data point. Common distance metrics include:
    - **Euclidean Distance:** d(p,q)=∑i=1n(pi−qi)2d(p, q) = \sqrt{\sum_{i=1}^n (p_i - q_i)^2}d(p,q)=∑i=1n(pi−qi)2
    - **Manhattan Distance:** d(p,q)=∑i=1n∣pi−qi∣d(p, q) = \sum_{i=1}^n |p_i - q_i|d(p,q)=∑i=1n∣pi−qi∣
    - **Minkowski Distance:** Generalization of Euclidean and Manhattan distances.
    - **Cosine Similarity:** d(p,q)=1−p⋅q∥p∥ ∥q∥d(p, q) = 1 - \frac{p \cdot q}{\|p\| \|q\|}d(p,q)=1−∥p∥ ∥q∥p⋅q for text data or high-dimensional spaces.
3. **Parameter k:** The number of nearest neighbors (k) is a critical hyperparameter. It controls the algorithm's complexity and bias-variance trade-off:
    - **Small k:** Leads to high variance and low bias (overfitting).
    - **Large k:** Leads to high bias and low variance (underfitting).
4. **Weighted Voting:** In some variants of KNN, neighbors can be weighted differently, typically inversely proportional to their distance from the query point.

### Steps of the KNN Algorithm:

**1. Load the Data:**

- Import the dataset and split it into training and test sets.

```
from sklearn.model_selection import train_test_split
```

```
from sklearn.datasets import load_iris


data = load_iris()

X_train, X_test, y_train, y_test = train_test_split(data.data,
data.target, test_size=0.3, random_state=42)
```

**2. Choose the Value of k:**

- Decide on the number of neighbors to consider.

**3. Calculate Distances:**

- Compute the distance between the query instance and all training samples.

```
from sklearn.neighbors import KNeighborsClassifier


knn = KNeighborsClassifier(n_neighbors=3)

knn.fit(X_train, y_train)
```

**4. Identify Nearest Neighbors:**

- Sort the distances and determine the k-nearest instances.

```
neighbors = knn.kneighbors([X_test[0]], n_neighbors=3)
```

**5. Voting Mechanism:**

- For classification, assign the class label by majority vote from the nearest neighbors.
- For regression, take the average of the nearest neighbors' values.

**6. Make Predictions:**

- Predict the class label or value for the test instance.

```
predictions = knn.predict(X_test)
```

**7. Evaluate the Model:**

- Assess the model's performance using appropriate metrics.

```
from sklearn.metrics import accuracy_score


accuracy = accuracy_score(y_test, predictions)

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

### Advantages of KNN:

1. **Simplicity:** Easy to understand and implement.
2. **No Training Phase:** KNN is a lazy learner, meaning it doesn't require a training phase.
3. **Adaptability:** Can be used for both classification and regression problems.
4. **Non-Parametric:** Makes no assumptions about the underlying data distribution.

### Disadvantages of KNN:

1. **Computational Cost:** High computation time during prediction due to distance calculations.
2. **Memory Usage:** Requires storing the entire training dataset.
3. **Sensitivity to Noise:** Sensitive to irrelevant features and the scale of data.
4. **Curse of Dimensionality:** Performance degrades with increasing dimensions.

### Best Practices:

**Feature Scaling:** Normalize or standardize features to ensure that all features contribute equally to the distance calculation.

```
from sklearn.preprocessing import StandardScaler


scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)
```

```
X_test = scaler.transform(X_test)
```

1.

**Optimal k:** Use techniques such as cross-validation to determine the best value of k.

```
from sklearn.model_selection import GridSearchCV


param_grid = {'n_neighbors': np.arange(1, 25)}

knn = KNeighborsClassifier()

knn_cv = GridSearchCV(knn, param_grid, cv=5)

knn_cv.fit(X_train, y_train)

print(knn_cv.best_params_)
```

**Distance Metric:** Choose an appropriate distance metric based on the nature of the data.

**Example Workflow:**

```
import numpy as np

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy_score


# Load data

data = load_iris()

X = data.data

y = data.target
```

```
# Split data

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)


# Feature scaling

scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)

X_test = scaler.transform(X_test)


# Initialize and train KNN

knn = KNeighborsClassifier(n_neighbors=5)

knn.fit(X_train, y_train)


# Make predictions

y_pred = knn.predict(X_test)


# Evaluate model

accuracy = accuracy_score(y_test, y_pred)

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