Introduction to Python and Data Science

Hariharan V M

What is Data Science?

- Data Science is the process of extracting insights and knowledge from data.
- It involves various types of problems such as prediction, classification, clustering, etc.

Regression Problems

- Objective: Predict a continuous value.
- Example Use Cases:
  - Predicting house prices.
  - Forecasting weather.
  - Estimating stock values.

Classification Problems

- Objective: Assign categories to data points.
- Example Use Cases:
  - Spam detection.
  - Disease diagnosis.
  - Fraud detection.

Clustering Problems

- Objective: Group similar data points together.
- Example Use Cases:
    - Customer segmentation.
    - Market research.
    - Document clustering.

Time Series Problems

- Objective: Predict future values based on historical data.
- Example Use Cases:
  - Stock market prediction.
  - Sales forecasting.
  - Energy consumption prediction.

Anomaly Detection

- Objective: Identify unusual patterns or outliers.
- Example Use Cases:
  - Fraud detection.
  - Network security.
  - Equipment failure prediction.

Definition of Data Manipulation

Data manipulation is defined as the process of organizing and transforming data to make it appropriate, consistent, and valuable for downstream analysis such as data analytics and machine learning.

Why is Data Manipulation Important?

- Ensures data clarity and readability.
- Improves the reliability and validity of analytics results.
- Crucial for effective supply chain decision making.

Data Appropriateness

- Selecting appropriate data from various sources for different purposes.
- Saves time and improves the accuracy of analytics models.

Data Consistency

- Ensures readability and efficiency.
- Unifies data from various sources, making it easier to organize and store.

Data Exploitation

- Enables analysts to edit, delete, merge, and combine data.
- Maximizes the value gained from available data for analytics purposes.

Four-Step Procedure for Data Manipulation

1. **Step 1: Data Collection** - Gather relevant data from various sources.
2. **Step 2: Data Cleaning** - Handle missing values, outliers, and inconsistencies.
3. **Step 3: Data Transformation** - Convert data into a suitable format for analysis.
4. **Step 4: Data Analysis** - Apply analytics models and interpret the results.

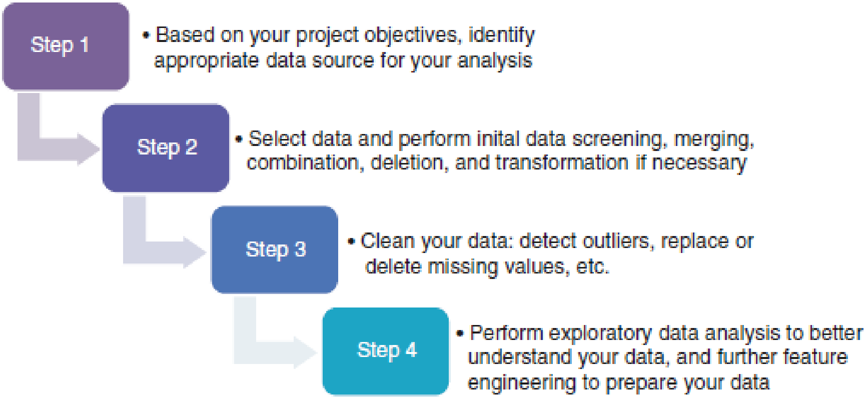
Iterative Data Manipulation Process

- In real scenarios, you may need to go back and forth between steps.
- For instance, if your data is clean, you may skip Step 3 and move to Step 4.
- If the data is inadequate, return to Step 2 to gather additional data.

Using Pandas for Data Manipulation

- Pandas is a powerful Python library for data manipulation and analysis.
- The following sections explore various data manipulation techniques using Pandas.
- Note: Familiarity with Pandas basics is recommended.

Four Steps for Data Manipulation



Step 1

- Based on your project objectives, identify appropriate data source for your analysis

Step 2

- Select data and perform inital data screening, merging, combination, deletion, and transformation if necessary

Step 3

- Clean your data: detect outliers, replace or delete missing values, etc.

Step 4

- Perform exploratory data analysis to better understand your data, and further feature engineering to prepare your data

Figure: Four Step Process for Data Manipulation

Python - Overview Example

```
class Stadium(object):

        def __init__(self, year, location, length, width, capacity, ticketprice):  # atributes of Stadium
                self.year = year
                self.location = location
                self.length = length
                self.width = width
                self.capacity = capacity
                self.ticketprice = ticketprice

        # define methods for Stadium
        def get_size(self):
                return self.length * self.width

        def get_revenue(self):
                return self.capacity * self.ticketprice
```

Numpy

```
>>> import numpy as np
>>> dir(np)
>>> len(dir(np))
534
```

Arrays and Operations

```
>>> numpy.array(object, dtype=None, copy=True, order='K'
    , subok=False, ndmin=0)
```

Parameter	Description	Default Value
object	Input data (list, tuple, etc.)	Required
dtype	Data type of array elements	None
copy	If True, copies the input data	True
order	Memory layout (' C', ' F ' , or $'K'$ ')	' K'
subok	If True, uses the subclass of object	False
ndmin	Minimum number of dimensions	0

'C' - C Order (Row Major Order) ; 'F' - Fortron-Order ; 'K' - Keep the Order

Array manipulation

```python
# Creating a 1D array from a list
arr = np.array([1, 2, 3, 4, 5])

# Creating an array with float data type
arr = np.array([1, 2, 3], dtype=float)

# Creating a 2D array (matrix)
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])

# Creating an array with at least 3 dimensions
arr = np.array([1, 2, 3], ndmin=3)
```

Various Array Operations

```
# Creating an array of zeros
zeros_array = np.zeros((3, 3))

# Creating an array of ones
ones_array = np.ones((2, 4))

# Creating an array with a range of numbers
range_array = np.arange(0, 10, 2)

# Creating a random array
random_array = np.random.rand(3, 3)

# Creating an identity matrix
identity_matrix = np.eye(3)
```

Manipulated Arrays

```python
arr = np.array([1, 2, 3, 4, 5, 6])
reshaped_arr = arr.reshape(2, 3)

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Addition
print(arr1 + arr2)  # Output: [5 7 9]

# Multiplication
print(arr1 * arr2)  # Output: [ 4 10 18]
```

Matrix Multiplication using @ Operator

```python
import numpy as np

# Creating two 2D arrays (matrices)
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

# Matrix multiplication
result = A @ B
print(result)
```

Matrix Multiplication using matmul function and Determinant

```python
import numpy as np

# Creating two 2D arrays (matrices)
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

# Matrix multiplication using np.matmul
result = np.matmul(A, B)
```

Determinant Syntax:

```python
det = np.linalg.det(matrix)
```

Eigen Value and Vector

Syntax:

```python
eigenvalues, eigenvectors = np.linalg.eig(matrix)
```

```python
import numpy as np

# Creating a matrix with complex eigenvalues
A = np.array([[0, -1],
[1,  0]])

# Calculating eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(A)

# Output results
print("Eigenvalues:")
print(eigenvalues)

print("\nEigenvectors:")
print(eigenvectors)
```

Linear Algebra in Numpy

```
>>> from numpy import linalg as lg
>>> dir(lg)
['LinAlgError', '__all__', '__builtins__', '__cached__',
    '__doc__', '__file__', '__loader__', '__name__', '
    __package__', '__path__', '__spec__', '_linalg', '
    _umath_linalg', 'cholesky', 'cond', 'cross', 'det', '
    diagonal', 'eig', 'eigh', 'eigvals', 'eigvalsh', 'inv
    ', 'linalg', 'lstsq', 'matmul', 'matrix_norm', '
    matrix_power', 'matrix_rank', 'matrix_transpose', '
    multi_dot', 'norm', 'outer', 'pinv', 'qr', 'slogdet',
     'solve', 'svd', 'svdvals', 'tensordot', 'tensorinv',
     'tensorsolve', 'test', 'trace', 'vecdot', '
    vector_norm']
```

Linear Algebra - Contd

Function	**Definition**
cholesky	Computes the Cholesky decomposition of a Hermitian, positive-definite matrix.
cross	Computes the cross product of two vectors in 3-dimensional space.
det	Computes the determinant of a square matrix.
diagonal	Returns the specified diagonals of an array or matrix.
eig	Computes the eigenvalues and right eigenvectors of a square matrix.
eigh	Computes the eigenvalues and eigenvectors of a Hermitian or symmetric matrix.
eigvals	Computes only the eigenvalues of a square matrix.
eigvalsh	Computes only the eigenvalues of a Hermitian or symmetric matrix.
inv	Computes the (multiplicative) inverse of a square matrix.
lstsq	Computes the least-squares solution to a linear matrix equation.
matmul	Performs matrix multiplication.

Linear Algebra - Contd

Function	**Definition**
matrix_norm	Computes a matrix norm.
matrix_power	Raises a square matrix to a given integer power.
matrix_rank	Computes the rank of a matrix.
matrix_transpose	Returns the transpose of a matrix.
multi_dot	Efficiently computes the dot product of two or more matrices.
norm	Computes the norm of a vector or matrix.
solve	Solves a linear matrix equation or system of linear scalar equations.
svd	Computes the Singular Value Decomposition (SVD) of a matrix.
svdvals	Computes the singular values of a matrix.
trace	Returns the sum of the elements on the main diagonal of a matrix.
vecdot	Computes the dot product of two vectors.
vector_norm	Computes the norm of a vector.

Singular Value Decomposition

Syntax:

```
U, S, V = numpy.linalg.svd(a, full_matrices=True,
    compute_uv=True, hermitian=False)
```

Code:

```
import numpy as np
# Define a matrix
A = np.array([[1, 2], [3, 4], [5, 6]])
# Compute SVD
U, S, Vh = np.linalg.svd(A)
# Print the results
print("U:\n", U)
print("Singular values:\n", S)
print("Vh:\n", Vh)
```

Data I/O Methods in Pandas

Data Type	**Reader**	**Writer**
CSV	read_csv	to_csv
JSON	read_json	to_json
HTML	read_html	to_html
Local Clipboard	read_clipboard	to_clipboard
MS Excel	read_excel	to_excel
HDF5 Format	read_hdf	to_hdf
Feather Format	read_feather	to_feather
Parquet Format	read_parquet	to_parquet
Msgpack	read_msgpack	to_msgpack
Stata	read_stata	to_stata
Python Pickle Format	read_pickle	to_pickle
SQL	read_sql	to_sql

Table: Data I/O Methods in Pandas