Machine Learning Lab

Week1:

Task: Data Wrangling changing data to meaningful outicomes

Required Python Libraries: numpy, pandas

Pandas is a powerful data manipulation and analysis library for Python. It provides data structures like series and dataframes to effectively easily clean, transform, and analyze large datasets and integrates seamlessly with other python libraries, such as numPy and matplotlib. It offers powerful functions for data transformation, aggregation, and visualization.

1. Pandas – Data Wrangling Made Easy

is an open-source library specifically designed for data manipulation and analysis. It provides two essential data structures: **Series** (**1-dimensional**) and **DataFrame** (**2-dimensional**), which make it easy to work with structured data, such as tables or CSV files.

- **Data Cleaning**: Pandas help in handling missing values, duplicates, and data transformations.
- **Data Exploration**: You can easily filter, sort, and group data to explore trends.
- **File Handling**: Pandas can read and write data from various file formats like CSV, Excel, SQL, and more.

importing Pandas

code:

```
import pandas as pd
```

- **import pandas**: Loads the Pandas library.
- **as pd**: Gives it the alias pd to make it easier to use.

Creating a Series

Code:

```
import pandas as pd
data = [10, 20, 30, 40]
v = pd.Series(data)
print(v)
```

pd.Series(data): Creates a one-dimensional labeled array (Series) from the data list.

Index: Automatically assigned as 0, 1, 2, etc.

Values: Are the items in the list [10, 20, 30, 40].

Creating a DataFrame

```
data = {
   'Name': ['Alice', 'Bob', 'Charlie'],
   'Age': [25, 30, 35],
   'City': ['New York', 'Los Angeles', 'Chicago']
}
df = pd.DataFrame(data)
print(df)
```

- **pd.DataFrame(data)**: Converts the dictionary data into a tabular (row and column) format.

- **Keys**: Become column names (Name, Age, City).
- **Values**: Corresponding rows.

Reading Data from Files

Code:

df = pd.read_csv('example.csv')

or

df=pd.read_csv("transactions.csv")

transactions.cav dataset download from this link:

https://github.com/ben519/DataWrangling/blob/master/Data/transactions.csv

- **pd.read_csv('filename.csv')**: Reads a CSV file and converts it into a DataFrame.
- Ensure that example.csv is in the same directory as your script or provide the full path.

Viewing the Data

One of the most used method for getting a quick overview of the DataFrame, is the **head()** method.

The **head()** method returns the headers and a specified number of rows, starting from the top.

Ex:

printing the first 10 rows of the DataFrame:

code:

```
import pandas as pd

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

```
print(df.head(10))
```

head(): Useful to check the structure of the data.

The **head()** method will return the first 5 rows.

Ex:2

Code:

```
print(df.tail())
```

- There is also a **tail()** method for viewing the *last* rows of the DataFrame.
- The **tail()** method returns the headers and a specified number of rows, starting from the bottom

Ex:3

Code:

```
print(df.info())
```

- **info()** that gives you more information about the data set.
- he **info()** method also tells us how many Non-Null values there are present in each column, and in our data set.

Ex:4

Code:

```
print(df.describe())
```

- **describe()**: Summarizes data with metrics like mean, count, min, max, etc.

ex:5

code:

```
print(df['Name'])
```

- Accessing a single column

Ex:5

Code:

```
print(df[['Name', 'City']])
```

- Accessing multiple columns

ex:5

code:

- Accessing rows by index

```
print(df.iloc[1])
```

- Access the second row

❖ **df['column']**: Retrieves a specific column.

❖ **df[['col1', 'col2']]**: Retrieves multiple columns.

❖ **df.iloc[row_index]**: Accesses a row by its position.

Filtering Data

Code:

```
filtered_df = df[df['Age'] > 30]
print(filtered_df)
```

- **Condition (df['Age'] > 30)**: Filters rows where the Age column has values greater than 30.
- **Result**: Returns a new DataFrame with only matching rows.

Renaming Columns

Code:

df.rename(columns={'Name': 'Full Name', 'Age': 'Years'}, inplace=True) print(df)

- **rename(columns={old: new})**: Renames columns in the DataFrame.
- **inplace=True**: Updates the DataFrame directly without needing to assign it to a new variable.

Ex:

Code:

import pandas as pd

df = pd.read_csv("transactions.csv")

df.rename(columns={"Quantity": "Quant"}, inplace=True)

print(df.head())

df.rename(columns={...}):

- This method is used to rename the columns of a DataFrame.
- The columns parameter accepts a dictionary where the keys are the current column names, and the values are the new names.

Handling Missing Data

- Check for missing values

```
print(df.isnull().sum())
```

- Fill missing values in 'Age' column

```
df['Age'].fillna(df['Age'].mean(), inplace=True)
```

- Drop rows with missing values

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

- **isnull()**: Checks for NaN (missing) values.
- **fillna()**: Replaces missing values with a specified value (e.g., mean of the column).
- **dropna()**: Removes rows with any missing values.

The rows of transactions by Quantity ascending, TransactionDate descending

Code:

```
import pandas as pd

df = pd.read_csv("transactions.csv")

df_sorted = df.sort_values(by=["Quantity", "TransactionDate"], ascending=[True, False])

print(df_sorted.head())
```

sort_values(by=["Quantity", "TransactionDate"]):

- The by parameter specifies the columns to sort by.
- Here, the DataFrame is sorted by Quantity first and then by TransactionDate.

ascending=[True, False]:

- This specifies the sorting order for each column.
- True for Quantity means ascending order.
- False for TransactionDate means descending order.

➢

Assign the result:

- ➢ The sort_values() method returns a new DataFrame with the specified sorting applied.
- ➢ Assign it to a new variable (df_sorted) or overwrite the original DataFrame (df).

Print the sorted DataFrame:

- Use df_sorted.head() to view the first few rows of the sorted data.

Subsetting data

- Example: Subset rows where Quantity > 1:
- ```python
- subset = df[df['Quantity'] > 1]
- print(subset)

You can combine conditions using `&` and `|`

Ex:1

Code:

Subset rows 1, 3, and 6

```
subset_rows = df.iloc[[0, 2, 5]]
print(subset_rows)
```

iloc Method:

- The iloc method is used for integer-location-based indexing.
- The rows are specified as a list of indices: [0, 2, 5].

Subset rows where an external array, foo, is True

Code:

```
foo = [True, False, True, False]  # Example of a boolean array
subset_rows = df[foo]
print(subset_rows)
```

Boolean Mask (foo):

- The foo array acts as a filter mask for the rows.
- It should have the same length as the DataFrame (number of rows), and each value in foo should correspond to whether the row is kept (True) or excluded (False).

Subsetting with the mask:

- By passing foo directly inside df[foo], you're selecting rows where the corresponding foo value is True.

Subset by columns TransactionID and TransactionDate with logical operator

Code:

```
subset = df[df['Quantity'] > 1][['TransactionID', 'TransactionDate']]
print(subset)
```

- df['Quantity'] > 1: This is the condition to filter rows where Quantity is greater than 1.
- [['TransactionID', 'TransactionDate']]: This selects the TransactionID and TransactionDate columns after applying the condition.

Subset columns by a variable list of column names

Code:

```
columns_to_select = ['TransactionID', 'TransactionDate']
subset_columns = df[columns_to_select]
print(subset_columns)
```

- **columns_to_select**: This variable holds the list of column names you want to include.
- **df[columns_to_select]:** This selects the columns from the DataFrame as specified in the list.

Subset columns excluding a variable list of column names

Code:

```
columns_to_exclude = ['Quantity', 'ProductID']
subset_columns_excluded = df[[col for col in df.columns if col not in columns_to_exclude]]
print(subset_columns_excluded)
```

- **columns_to_exclude:** This variable contains the list of column names that should be excluded from the selection.
- **df[[col for col in df.columns if col not in columns_to_exclude]]:** This list comprehension iterates over all column names and includes only those that are not in the **columns_to_exclude list**.

Convert the TransactionDate column to type Date

To convert a column in a pandas DataFrame to the datetime type, you can use the **pd.to_datetime()** function.

Code:

```
df['TransactionDate'] = pd.to_datetime(df['TransactionDate'])
print(df.dtypes)
```

- **pd.to_datetime()**: This function converts a given column (here, TransactionDate) into the datetime type.
- This allows you to perform date-based operations like filtering, sorting, and extracting date parts (e.g., year, month, day).

Insert a new column, Foo = UserID + ProductID

Code:

```
df['Foo'] = df['UserID'] + df['ProductID']
print(df.head())
```

- **df['UserID'] + df['ProductID']**: This adds the values of the UserID and ProductID columns for each row.
- **df['Foo'] =**: This creates a new column called Foo and assigns the result of the sum to it.

Inserting a New Row

Code:

```
df.loc[len(df)] = [new_user_id, new_product_id, new_quantity, new_transaction_date]
```

- **df.loc[len(df)]:** This adds a new row at the end of the DataFrame (because len(df) gives the next available index).

- The **list [new_user_id, new_product_id, new_quantity, new_transaction_date]** contains the values for the new row in the same order as the columns.

Updating Existing Values

Code:

```
df.loc[df['UserID'] == 2, 'Quantity'] = new_quantity_value
```

- df.loc[df['UserID'] == 2]: Selects rows where the UserID is 2.
- ['Quantity']: Specifies the column to update.
- new_quantity_value: The new value you want to set.

Inserting a New Column

Code:

```
df['Foo'] = df['UserID'] + df['ProductID']
```

- This directly assigns the result of **UserID + ProductID** to a new column Foo.

Modifying data frames

- Example: Adding a new column:
- ```python
- df['NewColumn'] = df['Quantity'] * 2
- print(df.head())

You can also update values directly.

Handling dates

- Convert the TransactionDate column to datetime:
- ```python
- df['TransactionDate'] = pd.to_datetime(df['TransactionDate'])
- ```
- This allows easy manipulation of dates.

Exercise questions:

Required Python Libraries: numpy, pandas

Problems:

Import Dataset [transactions.csv]

1. Print Summary of transaction data set
2. Print Numbers of Attributes
3. Print Numbers of Records
4. Get the row names
5. Get the column names
6. View top 10 Records
7. Change the name of column " Quantity" to "Quant"
8. Change the name of columns ProductID and UserID

to PID and UID respectively

9. Order the rows of transactions by TransactionId descending, if ascending then ascending=True

10. Order the rows of transactions by Quantity ascending, TransactionDate descending

11. Set the column order of Transactions as ProductID, Quantity, TransactionDate, TransactionID, UserID

12. Make UserID the first column of transactions

13. Extracting arrays from a Data Frame. Get the 2nd column

14. Get the ProductID Array

15. Get the ProductId array using the following variable

16.Row subsetting, subset rows 1,3 and 6

17. subset rows excluding 1,3 and 6

18. Subset the first three rows

19. Subset the last 2 rows

20. Subset rows excluding the last 2 rows

21. Subset rows excluding the first 3 rows

22. Subset rows where Quantity>1

23. Subset rows where UserID=2

24. Subset rows where Quantity>1 and UserID=2

25. Subset rows where Quantity + UserID is >3

26. Subset rows where an external array, foo, is True

27. Subset rows where an external array, bar, is positive

28. Subset rows where foo is TRUE or bar is negative

29. Subset the rows where foo is not TRUE and bar is not negative

30. Subset by columns 1 and 3

31. Subset by columns TransactionID and TransactionDate

32. Subset by columns TransactionID and TransactionDate with logical operator

33. Subset columns by a variable list of column names

34. Subset columns excluding a variable list of column names

35. Inserting and updating values

36. Convert the TransactionDate column to type Date

37. Insert a new column, Foo=UserID+ProductID