Pandas

→ Essentials

- Install pandas: ! pip install pandas
- Import pandas : import pandas as pd
- Check version : print(pd.__version__)

→ Importing Files

- csv : df = pd.read_csv("File.csv")
- text with separators : pd.read_csv("File.txt", sep="\t")
  - "\s" : space
  - "\t" : tab
  - "," : comma
- excel :
  - ↳ Single Sheet : pd.read_excel("File.xlsx")
  - ↳ multiple " : pd.read_excel("file.xlsx", sheet_name=0)

sheet_name (string) / sheet_pos (int)

- JSON : pd.read_json("file.json")

→ Exporting Files

index=True → writing the dataframe's index

- csv : pd.to_csv("output.csv", index=False)
- excel : pd.to_excel("output.xlsx", index=False)

→ Data Exploration

- df.head()
- df.head(12) ← n
  - ↳ First 5 (default) or n rows
- df.tail()
- df.tail(n)
  - ↳ last 5 (default) or n rows

- df.shape (rows, columns)
- df.shape[0] nb. of rows
- df.shape[1] nb. of columns

- df.columns columns name
- df.info()
  - ↳ data types, memory, non-null count

- df.describe() summary stats for numeric columns (count, mean, std, min, max, pcts)
- df.describe().T transposed
- df2 = df : a reference to the same df (any change in df2 will affect df and vice versa)
- df2 = df.copy() : new, independent copy

→ Dealing with missing data

. Detect missing values:

- df.isnull().sum() missing values per column
- df.isnull().sum().sum() total missing value in dataset

. Missing values solutions

↳ Option 1 : Drop missing values

Useful when we have plenty of data and losing a small portion won't impact the analysis

- df.dropna() drop rows with any missing values
- df.dropna(axis=1) drop columns with any missing value

inplace=True modifies the original df directly

↳ Option 2 : Replacing missing values

Replacing with a summary statistic or a specific value

- df.fillna(value) : Fill with specific value
- df.fillna(df.mean()) : Fill all NaN values in the df with the mean of each column
- df['col'].fillna(df['col'].mean()) : Fill NaNs in a single column with the mean of that column

→ Dealing with Duplicate Data

- df.drop_duplicates()

→ Column Management

. Rename columns

- df.rename(columns={'old_name':'new_name'}, inplace=True)

axis=0 (rows) default

. Drop a column/s

- df.drop(['col1','col2'], axis=1) multiple
- df.drop('col', axis=1) single

axis=1 (columns)

. Creat new columns : we can create new columns from existing one. for example:
- df['new_col'] = df['col1'] / df['col2']

→ Data Selection and Filtering

Dataframe 2D table (rows, cols)

. Column Selection
- df['column'] Single column → returns Series (1D)
- df[['col1', col2']] multiple " → returns dataframe (2D)

Series 1D column from df

. Row Selection
- df[df.index == 1] Single row by index
- df[df.index.isin(range(2, 10))] multiple rows by range
  included . excluded

Index: row or column label

. Conditional Filtering
- df[df.['col'] == value]
- df[df.['age'] <= 30] age is 30 or younger
- df[df['gender'] != 'Male'] gender is not male.
- df[df['city'].isin(['Beirut', 'Jnoub'])] city is either Beirut or Jnoub

- df[~df['status'].isin(['Inaclive', 'Banned'])]
  ↳ status is not Inactive or Banned

& : and
| : or

- df[(df['age']>18) & (df['country'] == 'lebanon')]
  ↳ multiple conditions with and

- df[df['score'].isna()] : where score is NaN

- df[df['title'].str.contains('AI')]
  ↳ where title contains the word AI

↳ We can also use .query() as an alternative syntax
- df.query("age>25 and gender=='Female'")

→ Data Analysis

• Summary Statistics

- df.mean() - df.median() - df.mode()

• Value Counts

- df['col'].value_counts() count unique values
- df['col'].value_counts(normalize = True) as proportion
- df.value_counts(subset = ['col1', 'col2']
  ↳ multi-columns count

• Grouping Data

- df.groupby('column').mean(): group by one column
- df.groupby(['col1', 'col2']).mean() : " " multiple columns

→ Data Manipulation

• Sorting

Single col - df.sort_values(by = "col", ascending = False, inplace = True)

Multiple col - df.sort_values(by = ["col1", "col2"], ascending = [False, True])

• Index Management

- df.reset_index(drop = True, inplace = True)
  ↳ reset index after Filtering / Sorting)

Outliers

→ Core Concepts

• What are Outliers

↳ Definition: Data points that are significantly different/far from other observations

↳ Impact: Can skew statistical analysis and machine learning models

• Why detect Outliers

→ Data Quality: Identify data entry errors

→ Statistical Accuracy: Prevent skewed results

→ Model Performance: Improve prediction accuracy

→ Business Insights: Find unusual patterns or anomalies

→ IQR Method (= Interquartile Range)

• Simple way to detect and handle outliers in numerical data

• Python Code

1) Calculate quartiles
```
Q1 = df['col'].quantile(0.25)   25th percentile
Q3 = df['col'].quantile(0.75)   75th percentile
```

2) Calculate IQR
```
IQR = Q3 - Q1
```

3) Calculate bounds
```
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
```

4) Find outliers
```
outliers = df[(df['col'] < lower_bound) | (df['col'] > upper_bound)]
```

5) Drop outliers
```
df_clean = df.drop(outliers.index)
```

→ Z-Score Method (Alternative to IQR)

```
From scipy import stats
# Calculate Z-scores
z_scores = np.abs(stats.zscore(df['col']))

# Find outliers (threshold = 3)
outliers = df[z_scores > 3]
```

→ Visualization Methods

. Boxplots

```
# Basic
plt.figure(figsize=(10, 6))
sns.boxplot(data=df)
plt.show()

# Horizontal boxplot
sns.boxplot(data=df, orient='h')

# Single column boxplot
sns.boxplot(y=df['column'])
```

Normalization/Standardization

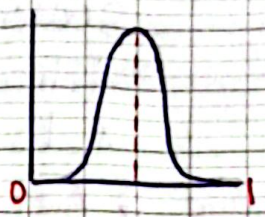
→ Core Concepts

• Feature Scaling

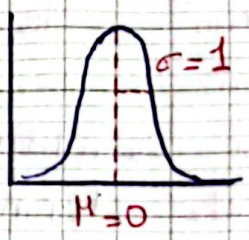
↳ Purpose: Transform features to similar scales
↳ Problem: Feature with larger values dominate smaller ones
↳ Solution: Scall all Features to comparable ranges

• Two Main Methods

1) Normalization (Min-Max):
Scales to range [0, 1]



2) Standardization (Z-Score)
Centers at mean = 0
and Standard deviation = 1



→ Normalization

• Types:

1) Min-Max Normalization
. rescale Features to a range of 0 to 1
Formula:

$$X_{new} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

2) Log Normalization
. apply a logarithmic transformation to Features

3) Decimal Scaling
adjust the sizes of the features by shifting the decimal point to make the values smaller

4) Mean Normalization (mean centering)
adjust Features by subtracting the avg from each value.

- When should we normalize data?
  - when the distribution of the data is unknown or does not allow a Gaussian distribution
  - with distance-base algorithm : K-NN, Clustering
  - Neural Network : prefer [0,1] range

- Python Code

```
From sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transmor(X_train)
X_test_scaled = scaler.transform(X_test)
```

  - Always fit on training data only and not on test data (causes dataleakage)

we can specify the feature_range as param in norm. but not in stand.

→ Standardization

- Formula: $X\_standardized = (X - \mu) / \sigma$
- When should we standardize data?
  - Gradient-based algo. : SVM, Logistic Regression
  - PCA / dimensionality reduction
  - Outliers present : More robust than normalization

- Python Code

```
From sklearn.preprocessing import StandardScaler
Scaler= StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

only .transform on test data

↳ Important Notes

1) Standardization is more robust to outliers
2) We should apply std / norm before feature engineering and after splitting the data
3) Normalization preserves the relative rlt. between values

Encoding / loc, iloc

→ Core Concepts

- Encoding = Converting categorical data into numerical format for ml algorithms
- Main Types:
  - ↳ Manual Mapping (ordinal data)
  - ↳ Label Encoding (Nominal data → single column)
  - ↳ One-Hot Encoding (Nominal data → multiple columns)

→ Basic Operation (not related to encoding)

- Creating dataframe from dictionary

```
dict = {...}
df = pd.DataFrame(dict)
```

- Essential Data Exploration

```
# Check data types
df.dtypes

# view unique values
df['col'].unique()
```

- Data Filtering using loc
  - ↳ loc (label_based indexing): selects data using row and column names
  - ↳ iloc (position_based indexing): uses numerical indices

i) loc:

- df.loc[3] Single row by label/index name
- df.loc['C345'] here the id column represent the row index
- df.loc[[1,4,7]] select multiple row by labels
- df.loc[2:5, ['Brand','Year']] Select rows and specific columns
  (2:5 → row index; ['Brand','Year'] → columns index)
- df.loc[(df['Year'] > 2012) & (df['Brand'] == 'Hyundai')]
  ↳ conditional row selection
- df.loc['C123':'C456'] From C123 to C456 both included

2) iloc

- df.iloc[0] row 0, all columns
- df.iloc[[0,3,4]] rows 0, 3, 4
- df.iloc[1:5, 0:3] here we exclude
  so → rows 1 to 4, col ob
- df.iloc[-1] last row
- df.iloc[::-1] reverse the df

→ Label Encoding

```
From SKlearn.preprocessing import LabelEncod
le = LabelEncoder()
encoded_values = le.fit_transform(df['col'])
df['encoded_col'] = encoded_values
```

Results

↳ create integers : 0, 1, 2, 3, ...
↳ Alphabelical ordering by default
↳ Problem: Algorithm might assume Hyundai > Mani
(if encoded as 0, 1, 2) > Tal

→ ONE-Hot Encoding

```
one_hot_data = pd.get_dummies(df, columns=['col'
```

↳ this give boolean

```
# Convert boolean to int (1: True, 0: False)
df = one_hot_data.astype(int)
```

Results

↳ Creat new binary columns
↳ Each row has 1 in one column, 0 in others
↳ Advantage : no artificial ordering

N.b: Drop original column after encoding

→ Manual Mapping

```
df['luxury'] = ['low', 'medium', 'high', 'low', 'high']
# Manual mapping with dictionary
df['luxury'] = df['luxury'].map({"low":1, "medium":2,
                                 "high":3})
```

↳ use when categorical feature have natural order/ranking

Feature Selection

→ Why Feature Selection

- Goal: Find the best set of features for optimal model performance

- Benefits:
  - ↳ Reduces model complexity
  - ↳ Improves generalization
  - ↳ Increases accuracy
  - ↳ Removes redundant / irrelevant features

→ Data Preparation (Always first step)

- we should first apply the essential steps: load data, describe, check missing value
- also an important step before feature selection is to separate X and y
  - ↳ `y = df['target_column']`
    `X = df.drop('target_column')`

→ Model-Based Feature Importance

A. Logistic Regression

```
From sklearn.linear_model import LogisticRegression
model = LogisticRegression
model.fit(X, y)

importance = model.coef_[0] # get coefficients
# Display Importance
for i,v in enumerate(importance):
    print('Feature: %0d, Score: %.5f' % (i,v))
plt.bar(range(len(importance)), importance)
plt.show()
```

B. RandomForest

```
From sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier()
model.fit(X,y)
importance = model.feature_importances  Built-in
↳ for display same as before
```

C. Decision Tree

```
From sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier()
model.fit(X,y)
importance = model.feature_importances_
```

→ Correlation Analysis

. Basic correlation matrix

```
matrix = df.corr()
# visualize with heatmap
plt.imshow(matrix, cmap="coolwarm")
plt.colorbar
plt.show()
```

. PCA correlation Graph

```
From mlxtend.plotting import plot_pca_correlation_graph
Feature_names = df.columns.tolist()
figure, corr_matrix = plot_pca_correlation_graph(df,
                                                 Feature_names
                                                 dimension
                                                 =(1,2)
```

→ Statistical Feature Selection

. SelectKBest

```
From sklearn.feature_selection import SelectKBest, f_classif

FS = SelecKBest (score_func= F_classif,
                 K = 6) select top 6
X_selected = FS.fit_transform (X, y)
# Get selected features name
Selected_F = X.columns [FS.get_support()]
X_selected = pd.DataFrame (X_selected,
                           columns = selected_F)
```

. Score functions

↳ for classification : F_classif , chi2

↳ for regression : F_regression

→ Removing features with low variance

```
From sklearn.feature_selection import VarianceThreshold
# Remove zero-variance features (default)
sel = VarianceThreshold ()
X_new = sel.fit_transform (X)

# Custom threshold                    for binary features
sel= VarianceThreshold (threshold= 0.8*(1-0.8))
X_new = sel.fit_transform (X)
```

→ PCA (Principal Component Analysis)

- Dimensionality Reduction technique that
  - ↳ convert original features into new uncorrelated features (principal components)
  - ↳ Keep the most important patterns (variance) in the data
  - ↳ Helps with visualization , noise reduction...

- Python Code :

```
From sklearn.decomposition import PCA
pca = PCA (n_components = 2)        # int : nb. of resulted columns
or pca = PCA (n_components = 0.95)  # dec<1 : amount of remaining information
X_pca = pca.fit_transform (X)
```

N.B. : unlike PCA , in feature selection we can't know how much info we are loosing

Classification Algorithm

→ Decision Tree Classifier

. Concept

↳ Tree-like model that makes decisions by splitting data based on feature value
↳ Each internal node represents a test on an attribute
↳ Each branch represents outcome of the test
↳ Each leaf node represents a class label (classification) or value (regression)
↳ Uses recursive binary splitting to create decision rules
↳ Can handle both numerical and categorical data
↳ Prone to overfitting but highly interpretable

. Implementation

```
from sklearn.tree import DecisionTreeClassifier
# Create classifier
clf = DecisionTreeClassifier()
# or with parameters
clf = DecisionTreeClassifier(criterion="entropy", max_depth=3)
# Train
clf = clf.fit(X_train, y_train)
# Predict
y_pred = clf.predict(X_test)
# Accuracy
accuracy = metrics.accuracy_score(y_test, y_pred)
# Precision
precision = metrics.precision_score(y_test, y_pred)
```

from sklearn import metrics

↳ Parameters

. Criterion = "entropy" or "gini" (default)
. max_depth : Limit tree depth to prevent overfitting

- Visualization

```
From sklearn.tree import plot_tree
plot_tree (clf)
```

→ Random Forest

- Concept
  - ↳ Ensemble method using multiple decision trees
  - ↳ Classification: majority vote from all trees
  - ↳ Regression: average of all tree predictions
  - ↳ Reduces overfitting compared to single decision trees

- Implementation

```
from sklearn.ensemble import RandomForestRegressor
# create model
rf = RandomForestRegressor(n_estimators = 1500,   # nb. of trees
                           random_state = 42)     # for reproducibility
# Train
rf.fit (X_train, y_train)
# predict
pred = rf.predict (X_test)
# Calculate errors
errors = abs (pred - y_test)
mae = np.means (errors)
```

→ K-Nearest Neighbors (K-NN)

- Concept
  - ↳ Non-parametric classification method
  - ↳ Classifies based on K closet training examples
  - ↳ Classification: plurality vote of neighbors
  - ↳ Regression: average of K nearest neighbors

- Implementation

```
From sklearn.neighbors import KNeighborsClassifier
Knn = KNeighborsClassifier (n_neighbors = 5)
Knn.fit (X_train, y_train)
predictions = knn.predict (X_test)
```

or

```
From sklearn.neighbors import NearestCentroid
clf = NearestCentroid
clf.fit (X_train, y_train)
clf.predict (X_test)
```

→ Naive Bayes

- Concept
  - ↳ Probabilistic classifier based on Baye's theorem
  - ↳ Asume strong independence between features
  - ↳ Highly scalable, linear parameters

- Implementation

```
From sklearn.naive_bayes import GaussianNB
nb = GaussianNB()
nb.fit (X_train, y_Train)
y_pred = nb.predict (x_test)
```

→ Support Vector Machines (SVM)

- Concept
  - ↳ Supervised learning of classification and regression
  - ↳ Maps data to maximize gap between categories

- Implementation

```
From sklearn.svm import SVC
sv = SVC()
sv.fit (X_train, y_train)
pred = sv.predict (x_test)
```

→ Logistic Regression

• Concept

↳ Measures relationship between categorical dependent variable and independent variables

↳ uses logistic function

• Implementation

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression(solver='liblinear')
model.fit(X_train, y_train)
pred = model.predict(X_test)
```

→ WorkFlow

```
# 1. get data
```

```
from sklearn.datasets
import load_iris
X, y = iris.data,
       iris.target
```

```
from google.colab import fi
uploaded = files.upload()
df = pd.read_csv('file.csv')
X = df.drop['outcome']
y = df['outcome']
```

From sklearn. model_selection import train_test_split

```
# 2. Split data
X_train, X_test, y_train, y_test = train_test_split(
                                   X, y, test_size=0
# 3. Create and train model
model = ClassifierName()
model.fit(X_train, y_train)

# 4. Predict
y_pred = model.predict(X_test)
```

5, Evaluation

```
from sklearn.metrics import accuracy_score,
                            precision_score,
                            confusion_matrix

accuracy = accuracy_score (y_test, y_pred)
precision = precision_score (y_test, y_pred)
cm = confusion_matrix (y_test, y_pred)

errors = abs (y_pred - y_test)
mae = round (np.means (errors), 2)
```

Association Rules

Installation

```
! pip install apyori
```

Imports

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
From apyori import apriori
```

Data load and preparation

```
data = pd.read_csv('File.csv', header=None)
# convert to transaction format
records []
for i in range (0, nbofTransactions):   # nbofTransactions = rows
    records.append ([str(data.values[i,j]) for j in range
                                        (0, nbof col)])
```

Apriori Algorithm

```
association_rules = apriori (records,
                             min_support = 0.0045,
                             min_confidence = 0.2,
                             min_lift = 3,
                             min_length = 2)

association_results = list(association_rules)
```

↳ parameters:

- min_support : 0.001 to 0.01 (0.1% to 1%): lower for rare items
- min_confidence : 0.1 to 0.8 (higher for stronger rules)
- min_lift : 1.0 to 5.0 (higher for stronger associations)
- min_length : 2 (minimum items in a rule)