Quick Navigation

Dependencies and Configuration

In [28]:
```
import random
from typing import List

import numpy as np
import pandas as pd
```

In [29]:
```
class global_config:

    # File Path
    raw_data = "../data/raw/data.csv"
    processed_data_stage_1 = "../data/processed/data_stage_1.csv"
    processed_data_stage_2 = "../data/processed/data_stage_2.csv"
    processed_data_stage_3 = "../data/processed/data_stage_3.csv"

    # Data Information
    target = ["diagnosis"]
    unwanted_cols = ["id", "Unnamed: 32"]

    # Plotting
    colors = ["#fe4a49", "#2ab7ca", "#fed766", "#59981A"]
    cmap_reversed = plt.cm.get_cmap('mako_r')

    # Seed Number
    seed = 1992

    # Cross Validation
    num_folds = 5
    cv_schema = "StratifiedKFold"
    split_size = {"train_size": 0.9, "test_size": 0.1}


def set_seeds(seed: int = 1234) -> None:
    """Set seeds for reproducibility."""
    np.random.seed(seed)
    random.seed(seed)
```

In [30]:
```
# set config
config = global_config

# set seeding for reproducibility
_ = set_seeds(seed = config.seed)
```

Stage 1: Preliminary Data Inspection and Cleaning

Load the dataset

In [31]:
```
df = pd.read_csv(config.raw_data)
```

A brief look at the dataset

- We will query the first five rows of the dataframe to get a feel on the dataset we are working on.
- We also call `df.info()` to see the data types of the columns, and to briefly check if there is any missing values in our data (more on that later).

```
#   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   diagnosis                569 non-null    int64
 1   radius_mean              569 non-null    float64
 2   texture_mean             569 non-null    float64
 3   perimeter_mean           569 non-null    float64
```

Importance of data types: We must be sharp and ensure that each column is indeed stored in their respective data types! In the real world, we may often query "dirty" data from say, the database, where numeric data are represented in string. It is now our duty to ensure sanity checks are in place!

In [32]:
```
display(df.head())
# display(df.info())
```

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	...	te
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	...	
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	...	
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	...	
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	...	
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	...	

5 rows × 33 columns

A brief overview tells us our data is alright! There is, however, a column which is unnamed and has no values. This can be of various data source issues, for now, we quickly check the definition given by the dataset from UCI's Breast Cancer Wisconsin (Diagnostic) Data Set and confirm that there should only be 32 columns. With this in mind, we can safely delete the column.

We also note that from the above, that the **id** column is the identifier for each patient. We will also drop this column as it holds no predictive power.

When can ID be important?
- We should try to question our move and justify it. In this dataset, we have to ensure that each **ID** is unique, if it is not, it may suggest that there are patient records with multiple observation, which is a violation of **i.i.d** assumption and we may take note when doing cross-validation, so as to avoid information leakage.
- Since the ID column is unique, we will delete it. We will keep this at the back of our mind in the event that we ever need them for feature engineering.

In [33]:
```
print(f"The ID column is unique : {df['id'].is_unique}")
```

```
The ID column is unique : True
```

Drop, drop, drop the columns!

Here we define a `drop_columns` function to drop the unwanted columns.

In [34]:
```
def drop_columns(df: pd.DataFrame, columns: List) -> pd.DataFrame:
    """[summary]

    Args:
        df (pd.DataFrame): [description]
        columns (List): [description]

    Returns:
        pd.DataFrame: [description]
    """

    df_copy = df.copy()
    df_copy = df_copy.drop(columns=columns, axis=1, inplace=False)
    return df_copy.reset_index(drop=True)
```

In [35]:
```
df = drop_columns(df, columns=config.unwanted_cols)
```

Data Types

Let us split the data types into a few unbrellas:

Categorical
- **diagnosis**: The target variable diagnosis, although represented as string in the dataframe, should be categorical! This is because machines do not really like working with "strings" and prefer your type to be of "numbers". We will map them to 0 and 1, representing benign and malignant respectively. Since the target variable is just two unique values, we can use a simple map from pandas to do the job.

In [36]:
```
class_dict = {"B" : 0, "M":1}
df['diagnosis'] = df['diagnosis'].map(class_dict)
```

We will make sure that our mapping is accurate by asserting the following.

In [37]:
```
assert df['diagnosis'].value_counts().to_dict()[0] == 357
assert df['diagnosis'].value_counts().to_dict()[1] == 212
```

Continuous
- **predictors**: A preliminary look seems to suggest all our predictors are continuous.

From the brief overview, there does not seem to be any Ordinal or Nominal Predictors. This suggest that we may not need to perform encoding in our preprocessing.

Summary Statistics

We will use a simple, yet powerful function call to check on the summary statistics of our dataframe. We note to the readers that there are much more powerful libraries like pandas-profiling to give us an even more thorough summary, but for our purpose, we will use the good ol' `df.describe()`.

In [38]:
```
display(df.describe(include='all'))
```

	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	symmetry
count	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.
mean	0.372583	14.127292	19.289649	91.969033	654.889104	0.096360	0.104341	0.088799	0.048919	0.
std	0.483918	3.524049	4.301036	24.298981	351.914129	0.014064	0.052813	0.079720	0.038803	0.
min	0.000000	6.981000	9.710000	43.790000	143.500000	0.052630	0.019380	0.000000	0.000000	0.
25%	0.000000	11.700000	16.170000	75.170000	420.300000	0.086370	0.064920	0.029560	0.020310	0.
50%	0.000000	13.370000	18.840000	86.240000	551.100000	0.095870	0.092630	0.061540	0.033500	0.
75%	1.000000	15.780000	21.800000	104.100000	782.700000	0.105300	0.130400	0.130700	0.074000	0.
max	1.000000	28.110000	39.280000	188.500000	2501.000000	0.163400	0.345400	0.426800	0.201200	0.

8 rows × 31 columns

The table does give us a good overview: for example, a brief glance give me the following observations:

1. The features **do not seem to be of the same scale**. This is going to be a problem as some models do not perform well if your features are not on the same scale. A prime example is a KNN model with Euclidean Distance as the distance metric, the difference in range of different features will be amplified with the squared term, and the feature with wider range will dominate the one with smaller range.
2. From our dataset it we see that **area_mean** is very large and there is likely to be a squared term (possibly from **radius_mean**), we can look into them later through EDA.

Humans are more visual and that is why we still need EDA later to capture our attention on any anomaly from the dataset, and of course, if the dataset has many columns, then this summary statistics may even clog your progress if you were to read it line by line.

Missing Data

Missing Alert! Although from our analysis, we did not see any missing data, it is always good to remind ourselves to check it. A simple function that does the job is as follows.

In [39]:
```
def report_missing(df: pd.DataFrame, columns: List) -> pd.DataFrame:
    """A function to check for missing data.

    Args:
        df (pd.DataFrame): [description]
        columns (List): [description]

    Returns:
        pd.DataFrame: [description]
    """
    missing_dict = {"missing num": [], "missing percentage": []}
    for col in columns:
        num_missing = df[col].isnull().sum()
        percentage_missing = num_missing / len(df)
        missing_dict["missing num"].append(num_missing)
        missing_dict["missing percentage"].append(percentage_missing)

    missing_data_df = pd.DataFrame(index=columns, data=missing_dict)

    return missing_data_df
```

In [40]:
```
missing_df = report_missing(df, columns=df.columns)
display(missing_df.head())
```

	missing num	missing percentage
diagnosis	0	0.0
radius_mean	0	0.0
texture_mean	0	0.0
perimeter_mean	0	0.0
area_mean	0	0.0

Save Data

After Stage 1 is done, we saved the data to our processed folder, we name it `processed_data_stage_1.csv`, indicating that the data is processed after stage 1.

In [41]:
```
df.to_csv(config.processed_data_stage_1, index=False)
```