Quick Navigation

Dependencies and Configuration

In [1]:
```
import random
from collections import defaultdict
from typing import Dict, List, Union

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn import (base, decomposition, linear_model, manifold, metrics,
                     preprocessing)

from statsmodels.stats.outliers_influence import variance_inflation_factor
```

In [2]:
```
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 [3]:
```
config = global_config

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

# read data
df = pd.read_csv(config.processed_data_stage_2)
```

Stage 3: Feature Engineering/Feature Selection

Disclaimer! We are fully aware that oftentimes practitioner may accidentally cause data leakage during preprocessing, for example, a subtle yet often mistake is to standardize the whole dataset prior to splitting, or performing feature selection prior to modelling using the information of our response/target variable. However, we can still screen predictors for multicollinearity during EDA phase and have a good intuition on which predictors are highly correlated - subsequently, we will incorporate a feature selection technique in our modelling pipeline.

Multicollinearity and Feature Selection

Motivation: We need feature selection in certain problems for the following reasons:
- Well, one would definitely have heard of the dreaded curse of dimensionality in the journey of learning Machine Learning where having too many predictor/features can lead to overfitting; on the other hand, too many dimensions can cause distance between observations to appear equidistance from one another. observations become harder to cluster — believe it or not, too many dimensions causes every observation in your dataset to appear equidistant from all the others, thereby clogging the model's ability to cluster data points (imagine the horror if you use KNN on 1000 dimensions, all the points will be almost the same distance from each other, poor KNN).
- In case you have access to Google's GPU clusters, you likely want to train your model faster. Reducing the number predictors can aid this process.
- Reducing uninformative features may aid in model's performance, the idea is to remove unnecessary noise from the dataset.

Multi-Collinearity: Looking back at our dataset, it is clear to me that there are quite a number of features that are correlated with each other, causing multi-collinearity. Multi-Collinearity is an issue in the history of Linear Models, quoting the statement from Is there an intuitive explanation why multicollinearity is a problem in linear regression? > Consider the simplest case where Y is regressed against X and Z and where X and Z are highly positively correlated. Then the effect of X on Y is hard to distinguish from the effect of Z on Y because any increase in X tends to be associated with an increase in Z. We also note that multi-collinearity is not that big of a problem for non-parametric models such as Decision Tree or Random Forests, however, I will attempt to show that it is still best to avoid in this problem setting.

Alert! Alert! Alert! There are many methods to perform feature selection. Scikit-Learn offers some of the following:
- Univariate feature selection.
- Recursive feature elimination.
- Backward Elimination of features using Hypothesis Testing.

EMERGENCY! We need to be careful when selecting features before cross-validation. It is therefore, recommended to include feature selection in cross-validation to avoid any "bias" introduced before model selection phase! I decided to use the good old Variance Inflation Factor (VIF) as a way to reduce multicollinearity. Unfortunately, there is no out-of-the-box function to integrate into the `Pipeline` of scikit-learn. Thus, I heavily modified an existing code in order achieve what I want below.

A classical way to check for multicollinearity amongst predictors is to calculate the Variable Inflation Factor (VIF). It is simply done by regressing each predictor x_i against all other predictors $\mathrm{x}_j, j \neq i$. In other words, the VIF for a predictor variable i is given by:

$$\mathrm{VIF}_i = \frac{1}{1 - R_i^2}$$

where R_i^2 is, by definition, the proportion of the variation in the "dependent variable" x_i that is predictable from the indepedent predictors $\mathrm{x}_j, j \neq i$. Consequently, the higher the R_i^2 of a predictor, the higher the VIF, and this indicates there is linear dependence among predictors.

Using Statsmodels Variance Inflation Factor

Note that we need to perform scaling first before fitting our `ReduceVIF` to get the exact same result as the previous version. In this version, I manually added a hard threshold for the number of features remaining to be 15. This hard coded number can be turned into a parameter (hyperparameter) in our pipeline.

In [4]:
```
class ReduceVIF(base.BaseEstimator, base.TransformerMixin):
    """The base of the class structure is not implemented by me, however, I heavily modified the class such that it
 can
    take in numpy arrays and correctly implemented the fit and transform method.
    """

    def __init__(self, thresh=10):
        self.thresh = thresh
        self.feature_names_ = None
        self.predictor_cols = [
            "radius_mean",
            "texture_mean",
            "perimeter_mean",
            "area_mean",
            "smoothness_mean",
            "compactness_mean",
            "concavity_mean",
            "concave points_mean",
            "symmetry_mean",
            "fractal_dimension_mean",
            "radius_se",
            "texture_se",
            "perimeter_se",
            "area_se",
            "smoothness_se",
            "compactness_se",
            "concavity_se",
            "concave points_se",
            "symmetry_se",
            "fractal_dimension_se",
            "radius_worst",
            "texture_worst",
            "perimeter_worst",
            "area_worst",
            "smoothness_worst",
            "compactness_worst",
            "concavity_worst",
            "concave points_worst",
            "symmetry_worst",
            "fractal_dimension_worst",
        ]

    def reset(self):

        self.predictor_cols = [
            "radius_mean",
            "texture_mean",
            "perimeter_mean",
            "area_mean",
            "smoothness_mean",
            "compactness_mean",
            "concavity_mean",
            "concave points_mean",
            "symmetry_mean",
            "fractal_dimension_mean",
            "radius_se",
            "texture_se",
            "perimeter_se",
            "area_se",
            "smoothness_se",
            "compactness_se",
            "concavity_se",
            "concave points_se",
            "symmetry_se",
            "fractal_dimension_se",
            "radius_worst",
            "texture_worst",
            "perimeter_worst",
            "area_worst",
            "smoothness_worst",
            "compactness_worst",
            "concavity_worst",
            "concave points_worst",
            "symmetry_worst",
            "fractal_dimension_worst",
        ]

    def fit(self, X, y=None):
        print("ReduceVIF fit")
        tmp, self.predictor_cols = ReduceVIF.calculate_vif(X, self.predictor_cols, self.thresh)
        self.feature_names_ = self.predictor_cols  # save as an attribute to call later
        col_index = [self.predictor_cols.index(col_name) for col_name in self.predictor_cols]
        self.col_index = col_index
        self.reset()
        return self

    def transform(self, X, y=None):
        print("ReduceVIF transform")
        # columns = X.columns.tolist()
        # print(X.shape)
        return X[:, self.col_index]

    @staticmethod
    def calculate_vif(X: Union[np.ndarray, pd.DataFrame], columns: List[str], thresh: float = 10.0):
        """Implements a VIF function that recursively eliminates features.

        Args:
            X (Union[np.ndarray, pd.DataFrame]): [description]
            columns (List[str]): [description]
            thresh (float, optional): [description]. Defaults to 10.0.

        Returns:
            [type]: [description]
        """

        dropped = True
        count = 0
        while dropped and count <= 15:
            column_index = X.shape[1]
            predictor_cols = np.arange(X.shape[1])
            dropped = False
            vif = []
            for var in range(column_index):
                # print(predictor_cols.shape)
                vif.append(variance_inflation_factor(X[:, predictor_cols], var))

            max_vif = max(vif)
            if max_vif > thresh:
                maxloc = vif.index(max_vif)
                print(f"Dropping {maxloc} with vif={max_vif}")
                # X = X.drop([X.columns.tolist()[maxloc]], axis=1)
                X = np.delete(X, maxloc, axis=1)
                columns.pop(maxloc)
                dropped = True
                count += 1
        return X, columns
```

In [10]:
```
predictor_cols = df.columns[1:]
transformer = ReduceVIF()
scaler = preprocessing.StandardScaler()
X = scaler.fit_transform(df[predictor_cols])
# Only use 10 columns for speed in this example
X = transformer.fit_transform(X)

vif_df = pd.DataFrame({'Predictors':predictor_cols})
```

```
ReduceVIF fit
Dropping 0 with vif=3806.1152963979675
Dropping 19 with vif=616.3508614719424
Dropping 1 with vif=325.64131198187516
Dropping 19 with vif=123.25781086343038
Dropping 4 with vif=64.65479584770004
Dropping 7 with vif=35.61751844352034
Dropping 19 with vif=33.96063880508537
Dropping 20 with vif=30.596655364833975
Dropping 1 with vif=25.387829695531387
Dropping 2 with vif=18.843208489973282
Dropping 14 with vif=17.232376192128665
Dropping 7 with vif=16.333806476471736
Dropping 15 with vif=15.510661467365699
ReduceVIF transform
```

In [11]:
```
print(f"Remaining Features: {transformer.feature_names_}")
display(vif_df.head(10))
```

```
Remaining Features: ['texture_mean', 'smoothness_mean', 'concave points_mean', 'symmetry_mean', 'fractal_dimension_me
an', 'texture_se', 'perimeter_se', 'smoothness_se', 'compactness_se', 'concavity_se', 'concave points_se', 'symmetry_
se', 'fractal_dimension_se', 'area_worst', 'smoothness_worst', 'symmetry_worst', 'fractal_dimension_worst']
```

	Predictors
0	radius_mean
1	texture_mean
2	perimeter_mean
3	area_mean
4	smoothness_mean
5	compactness_mean
6	concavity_mean
7	concave points_mean
8	symmetry_mean
9	fractal_dimension_mean

Oh Dear, we have a Multicollinearity Problem

Using VIF in Modelling Pipeline: At this step, we are just showing how we can remove multicollinear features using VIF; but we will not remove them at this point in time. We will incorporate this feature selection technique in our Cross-Validation pipeline in order to avoid data leakage.

Save the Data

In [12]:
```
df.to_csv(config.processed_data_stage_3, index=False)
```