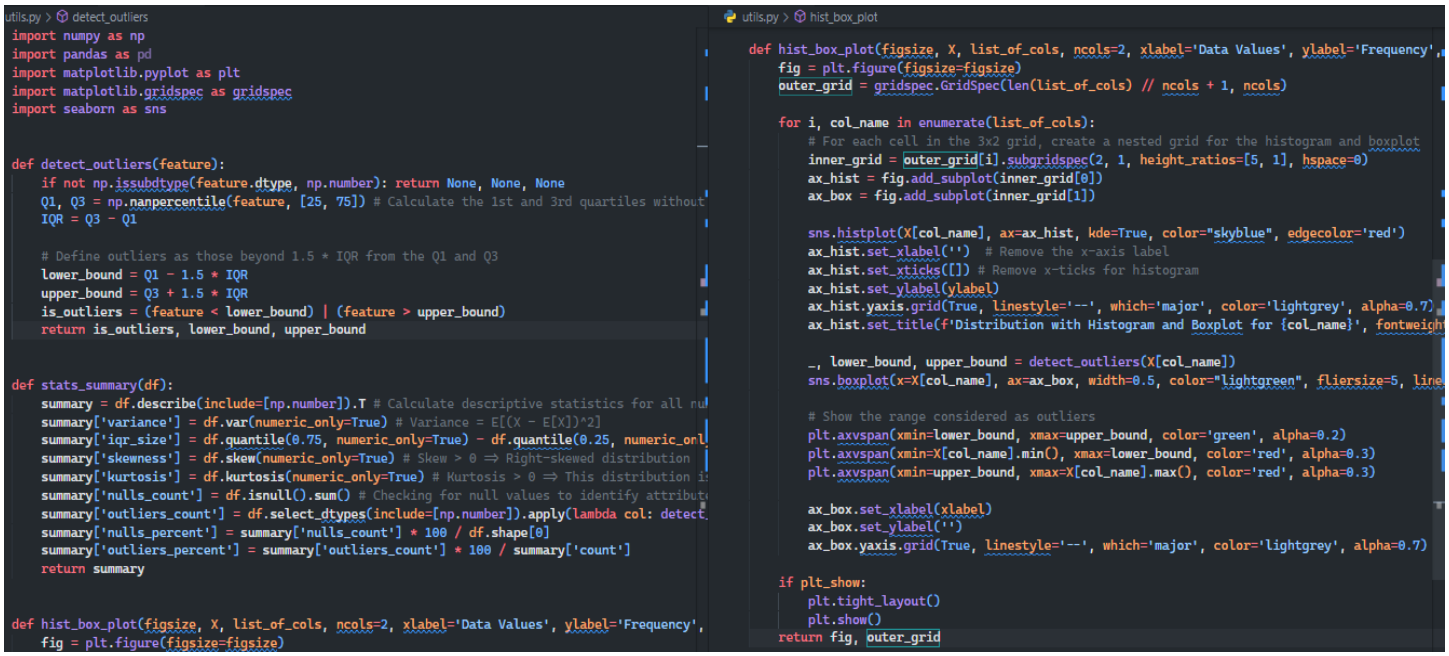
TABLE OF CONTENTS

I. Data Pre-processing and Transformation



```python
# utils.py > detect_outliers
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import seaborn as sns


def detect_outliers(feature):
    if not np.issubdtype(feature.dtype, np.number): return None, None, None
    Q1, Q3 = np.nanpercentile(feature, [25, 75]) # Calculate the 1st and 3rd quartiles without
    IQR = Q3 - Q1

    # Define outliers as those beyond 1.5 * IQR from the Q1 and Q3
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    is_outliers = (feature < lower_bound) | (feature > upper_bound)
    return is_outliers, lower_bound, upper_bound


def stats_summary(df):
    summary = df.describe(include=[np.number]).T # Calculate descriptive statistics for all nu
    summary['variance'] = df.var(numeric_only=True) # Variance = E[(X - E[X])^2]
    summary['iqr_size'] = df.quantile(0.75, numeric_only=True) - df.quantile(0.25, numeric_onl
    summary['skewness'] = df.skew(numeric_only=True) # Skew > 0 ⇒ Right-skewed distribution
    summary['kurtosis'] = df.kurtosis(numeric_only=True) # Kurtosis > 0 ⇒ This distribution i
    summary['nulls_count'] = df.isnull().sum() # Checking for null values to identify attribut
    summary['outliers_count'] = df.select_dtypes(include=[np.number]).apply(lambda col: detect
    summary['nulls_percent'] = summary['nulls_count'] * 100 / df.shape[0]
    summary['outliers_percent'] = summary['outliers_count'] * 100 / summary['count']
    return summary


def hist_box_plot(figsize, X, list_of_cols, ncols=2, xlabel='Data Values', ylabel='Frequency',
    fig = plt.figure(figsize=figsize)
```

```python
# utils.py > hist_box_plot
def hist_box_plot(figsize, X, list_of_cols, ncols=2, xlabel='Data Values', ylabel='Frequency',
    fig = plt.figure(figsize=figsize)
    outer_grid = gridspec.GridSpec(len(list_of_cols) // ncols + 1, ncols)

    for i, col_name in enumerate(list_of_cols):
        # For each cell in the 3x2 grid, create a nested grid for the histogram and boxplot
        inner_grid = outer_grid[i].subgridspec(2, 1, height_ratios=[5, 1], hspace=0)
        ax_hist = fig.add_subplot(inner_grid[0])
        ax_box = fig.add_subplot(inner_grid[1])

        sns.histplot(X[col_name], ax=ax_hist, kde=True, color="skyblue", edgecolor='red')
        ax_hist.set_xlabel('')  # Remove the x-axis label
        ax_hist.set_xticks([]) # Remove x-ticks for histogram
        ax_hist.set_ylabel(ylabel)
        ax_hist.yaxis.grid(True, linestyle='--', which='major', color='lightgrey', alpha=0.7)
        ax_hist.set_title(f'Distribution with Histogram and Boxplot for {col_name}', fontweight

        _, lower_bound, upper_bound = detect_outliers(X[col_name])
        sns.boxplot(x=X[col_name], ax=ax_box, width=0.5, color="lightgreen", fliersize=5, line

        # Show the range considered as outliers
        plt.axvspan(xmin=lower_bound, xmax=upper_bound, color='green', alpha=0.2)
        plt.axvspan(xmin=X[col_name].min(), xmax=lower_bound, color='red', alpha=0.3)
        plt.axvspan(xmin=upper_bound, xmax=X[col_name].max(), color='red', alpha=0.3)

        ax_box.set_xlabel(xlabel)
        ax_box.set_ylabel('')
        ax_box.yaxis.grid(True, linestyle='--', which='major', color='lightgrey', alpha=0.7)

    if plt_show:
        plt.tight_layout()
        plt.show()
    return fig, outer_grid
```

*This **utils.py** from Assessment 1 contains functions for in-depth statistical analysis and visualization.*

1. Initial Preprocess from Assessment 1

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from utils import *

missing_values = ['?', '.', '', ' ', '_', 'Na', 'NULL', 'null', 'not', 'Not', 'NaN', 'NA', '??', 'nan', 'inf']
raw_data = pd.read_csv('dataGaia_AB_train.csv', na_values=missing_values)
raw_data = raw_data.loc[:, ~raw_data.columns.str.contains('^Unnamed|Source')] # Drop 3 unnecessary columns
raw_data.set_index('ID', inplace=True)
```

```python
X = raw_data.drop(['SpType-ELS'], axis=1) # Set X to all columns except the target
Y = raw_data['SpType-ELS'].str.strip().str.upper() # Set Y to the target column
sp_type_counts = Y.value_counts()
print(f'{X.shape[1]} features and 1 target column:\n{X.columns}')
sp_type_counts
```

```
25 features and 1 target column:
Index(['RA_ICRS', 'DE_ICRS', 'Plx', 'PM', 'pmRA', 'pmDE', 'Gmag', 'e_Gmag',
       'BPmag', 'e_BPmag', 'RPmag', 'e_RPmag', 'GRVSmag', 'e_GRVSmag', 'BP-RP',
       'BP-G', 'G-RP', 'pscol', 'Teff', 'Dist', 'Rad', 'Lum-Flame',
       'Mass-Flame', 'Age-Flame', 'z-Flame'],
      dtype='object')

SpType-ELS
A    80088
B    68450
```

The dataset covers various astrometric and photometric properties of celestial objects from the Gaia mission with a large number of **148,538** entries, **25** features & **1** target. This does not include these following columns:

- **ID**: This will be set as **index** column for the raw dataset's DataFrame.
- **Unnamed: 0**: also appear to be index columns and redundant.
- **Source**: a unique identifier, does not contribute to any analysis.

The **Source** and **Unnamed: 0** are removed from the beginning as they are non-informative for predictive modeling. Additionally, based on the related initial exploration from **Assessment 1**, the target column **SpType-ELS**, which is categorical with 2 classes **A** & **B**, will be binarized into numerical format (0 & 1) by using **LabelEncoder**.

```
from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()
Y = label_encoder.fit_transform(Y)
pd.value_counts(Y)
```

✓ 0.5s

```
0    80088
1    68450
```

The classes are now easier to process for ML algorithms, with **80,088** instances of class **0 (A)** and **68,450** instances of class **1 (B)**.

2. Data Splitting

Most ML models require dividing the **train/validation/test sets**. In cases where the data is small, we can simply divide it into **training/validation sets**. The purpose of the **training set** is to train the model, so it needs to account for a large proportion to help the model learn to cover all cases of the data. The **validation set** is to re-evaluate the model to identify if *overfitting* and *underfitting* occur.

Ones might wonder if we already have a **validation set**, so why we need an additional **test set**? The **test set** is selected so that the distribution and characteristics are most similar to real data. This **set** aims to test the model's performance if deployed in production. Usually, the size of the **test set** is equal to that of the **validation**. To choose the best model, we will base on the evaluation results of the **test set**.

Here, I will split the data into **training & test set** before preprocessing to prevent **Data leakage**, which occurs when the information from **test set** is mistakenly used to scale or **"fit"** the transformations applied to the **training set**. This can give the model an unfair advantage, as it would have prior knowledge about the **test set**, leading to overly optimistic performance metrics that do not generalize well to new, unseen data.

By splitting first, any further transformations in subsequent steps like **imputation**, **PCA**, or **scaling** are **"fit"** only on the **training set**. Then, the same transformations are applied to the **test set** without re-fitting, thus mimicking the real-world scenario

where a model sees only new, unseen data. Lastly, the **validation set** will be further split from the **training set** during *cross-validation* processes in modelling phase.

```python
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2, random_state=42, stratify=Y)
Y_cnts = pd.value_counts(Y).to_list()
y_train_cnts = pd.value_counts(y_train).to_list()
y_test_cnts = pd.value_counts(y_test).to_list()

pd.DataFrame({
    'Original Data': [*Y_cnts, sum(Y_cnts), f'{Y_cnts[0] / Y_cnts[1]:.5f}'],
    'Training Set': [*y_train_cnts, sum(y_train_cnts), f'{y_train_cnts[0] / y_train_cnts[1]:.5f}'],
    'Test Set': [*y_test_cnts, sum(y_test_cnts), f'{y_test_cnts[0] / y_test_cnts[1]:.5f}'],
}, index=[*label_encoder.classes_, 'Total', 'Ratio']).T
```

	A	B	Total	Ratio
Original Data	80088	68450	148538	1.17002
Training Set	64070	54760	118830	1.17001
Test Set	16018	13690	29708	1.17005

Considering the complexity and the dataset size (**148,538** entries), an **80/20** split is chosen. This ratio provides a substantial amount of data for **training** to capture the underlying patterns while still retaining enough data for **testing**. The **stratify** parameter in **train_test_split** ensures that both **training & test sets** have same proportion of classes as in original data. This is important in cases where there is class *imbalance*. By setting **stratify=Y**, where **Y** is the target variable, this ensures each class's representation in both **training & testing sets** mirrors that in the complete dataset, leading to more reliable and robust model evaluation.

3. Handling Missing Values

```python
summary = stats_summary(raw_data)
summary[['nulls_count', 'nulls_percent']]\
    [summary.nulls_count > 0].sort_values('nulls_count', ascending=False)
```

	nulls_count	nulls_percent
pscol	143685	96.732823
GRVSmag	64054	43.122972
e_GRVSmag	64054	43.122972
Age-Flame	38016	25.593451
Mass-Flame	12551	8.449690
Lum-Flame	2995	2.016319
z-Flame	2995	2.016319

Proper handling of missing data can prevent models from learning inaccurate or skewed information. Therefore, I will conduct 2 methods, **dropping** and **imputing** to handle them. Given their importance in determining stellar characteristics and since **imputation** can introduce bias or alter the distribution, the decision is based on the proportion of missing data and the feature's potential importance to classification.

- Missing **> 30%**: might be best to remove, since excessively high missing likely offer little predictive value, making these features unreliable for prediction.
- Missing **< 30%**: using a model-based approach to predict missing values, **KNN imputation**, for a more accurate reflection of underlying patterns. It estimates the values based on the similarities with neighbors. This provides a more nuanced approach than simple median or mean strategies.

```
removed_columns = ['GRVSmag', 'e_GRVSmag', 'pscol']
X_train.drop(columns=removed_columns, inplace=True, errors='ignore')
X_test.drop(columns=removed_columns, inplace=True, errors='ignore')
X_train.columns
```

```
Index(['RA_ICRS', 'DE_ICRS', 'Plx', 'PM', 'pmRA', 'pmDE', 'Gmag', 'e_Gmag',
       'BPmag', 'e_BPmag', 'RPmag', 'e_RPmag', 'BP-RP', 'BP-G', 'G-RP', 'Teff',
       'Dist', 'Rad', 'Lum-Flame', 'Mass-Flame', 'Age-Flame', 'z-Flame'],
      dtype='object')
```

Due to high percentage of missing, **pscol** with extreme sparsity of **97%**, **GRVSmag** and **e_GRVSmag** with **~43%** are **dropped**. For the rest of columns, including those with low percentages like **Mass-Flame** (**~8%**), **Lum-Flame** and **z-Flame** (only **~2%**), **KNNImputer** is applied to all of them as these columns contain valuable information about physical characteristics which can be predictive of target classes. **Age-Flame** with **~25.6%**, this is substantial but not overwhelming percentage and as a potentially significant feature, **KNNImputer** is also used to estimate the values.

```
from sklearn.impute import KNNImputer
imputer = KNNImputer(n_neighbors=7) # Imputation for columns with missing values
imputer.set_output(transform='pandas')
X_train = imputer.fit_transform(X_train)
X_test = imputer.transform(X_test)
X_train.isnull().sum()
```

```
RA_ICRS       0
DE_ICRS       0
Plx           0
PM            0
pmRA          0
pmDE          0
Gmag          0
e_Gmag        0
BPmag         0
e_BPmag       0
RPmag         0
e_RPmag       0
BP-RP         0
BP-G          0
G-RP          0
Teff          0
Dist          0
Rad           0
Lum-Flame     0
Mass-Flame    0
Age-Flame     0
z-Flame       0
```

4. Handling Outliers

Outliers can affect model performance, especially for distance-based classifiers, like **SVM** or **KNN**. Given the nature of data (astronomical measurements), some features might naturally exhibit extreme values and high variance. For features where outliers are considerable and might impact the model disproportionately, techniques like **winsorization** (trim extreme values to a specified percentile) or **Robust Scaling** (scale data according to the *median* and the *quantile range*) could be appropriate to control extreme values without losing essential data.

```python
from scipy.stats.mstats import winsorize

def winsorize_outliers(X, limits=[0.05, 0.05]):
    X_winsorized = X.copy()
    before_winsorize = stats_summary(X_winsorized)[['outliers_count', 'outliers_percent']]
    before_winsorize = before_winsorize[before_winsorize.outliers_count > 0]

    cols_with_outliers = before_winsorize.index
    X_winsorized[cols_with_outliers] = X_winsorized[cols_with_outliers].apply(lambda col: winsorize(col, limits=limits))
    after_winsorize = stats_summary(X_winsorized.loc[:, cols_with_outliers])[['outliers_count', 'outliers_percent']]

    # Concatenate the 2 DataFrames for comparison
    outliers_summary = pd.concat([
        before_winsorize.rename(columns={'outliers_count': 'outliers_count_before', 'outliers_percent': 'outliers_percent_before'}),
        after_winsorize.rename(columns={'outliers_count': 'outliers_count_after', 'outliers_percent': 'outliers_percent_after'})
    ], axis=1).sort_values('outliers_percent_after', ascending=False)
    return X_winsorized, outliers_summary
```

Normally, **dropping** is a popular method for addressing outliers. However, as it involves removing outlier observations completely from the dataset, this process not only affects the specific variable (column) with the outlier but also eliminates these data points across all variables. This poses a risk: data points that are outliers in a variable might not be outliers in others, potentially leading to the unintended loss of valid data points (inliers). A way to mitigate this issue is by assigning *null* values to the outliers instead of removing them entirely, thus preserving the integrity of other data in the row. **Winsorization** even goes beyond this. Instead of eliminating extreme values, the values at both ends of the distribution are replaced with the nearest remaining values.

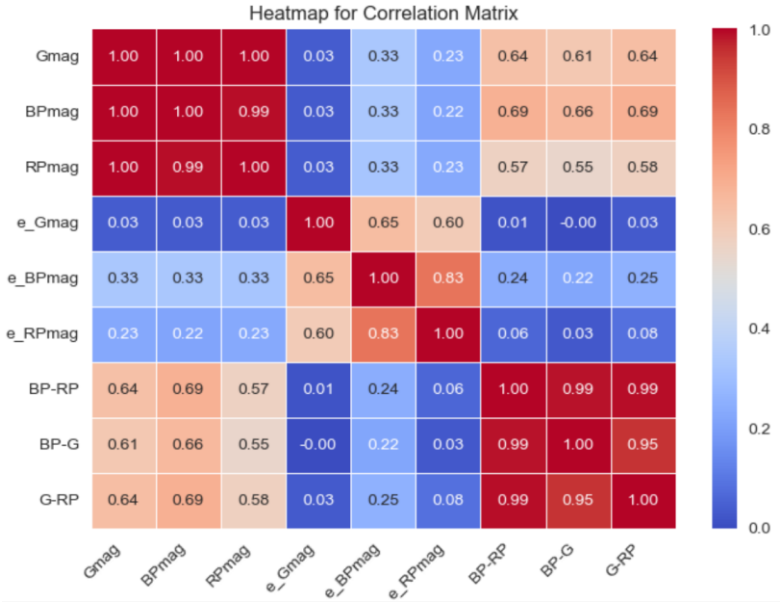
Here, I will first used the **1.5xIQR** rule to identify outliers within the dataset and then apply **winsorization** to reduce their influence by limiting them to the **5th** and **95th** percentiles: all values below **5th** percentile are set to the **5th** percentile value, while those above **95th** percentile are set to the **95th** percentile value. This helps in effectively reducing the impact of extreme outliers without removing particular features, which could distort the analysis and predictive modeling, making later training process more stable.

```
# Apply winsorizing to cap outliers within the 5th and 95th percentiles for selected columns
X_train, outliers_summary = winsorize_outliers(X_train, limits=[0.05, 0.05])
X_test, _ = winsorize_outliers(X_test, limits=[0.05, 0.05])
outliers_summary
```

	outliers_count_before	outliers_percent_before	outliers_count_after	outliers_percent_after
e_RPmag	18189	15.306741	18189	15.306741
e_BPmag	16939	14.254818	16939	14.254818
e_Gmag	14358	12.082807	14358	12.082807
Lum-Flame	13628	11.468484	13628	11.468484
DE_ICRS	9140	7.691660	9140	7.691660
PM	7878	6.629639	7878	6.629639
pmRA	13212	11.118404	7566	6.367079
pmDE	10590	8.911891	7555	6.357822
Mass-Flame	7039	5.923588	7039	5.923588
Plx	6748	5.678701	6672	5.614744
Teff	6235	5.246992	6235	5.246992
Gmag	393	0.330725	0	0.000000
BPmag	289	0.243205	0	0.000000
RPmag	557	0.468737	0	0.000000
BP-RP	956	0.804511	0	0.000000
BP-G	3447	2.900783	0	0.000000
G-RP	394	0.331566	0	0.000000
Dist	2272	1.911975	0	0.000000
Rad	5700	4.796768	0	0.000000
Age-Flame	663	0.557940	0	0.000000
z-Flame	5843	4.917108	0	0.000000

5. Dimensionality Reduction with PCA

PCA can reduce dimensionality of high-correlated features by transforming them into a new set of features, which are linear combinations of original ones. These new **principal components** will be chosen to explain the maximum variance within the data. This can simplify the model without losing significant information and reduce the noise by focusing on **higher variance components**. Fewer dimensions can also lead to faster training and potentially better performance by reducing the risk of *overfitting* as the model is less likely to learn noise in the data.



The corr matrix shows **3 groups** of high-correlated features (also discovered from **Assessment 1**): among magnitude measures (**Gmag**/**BPmag**/**RPmag**), their errors (**e_Gmag**/**e_BPmag**/**e_RPmag**), and between certain color indexes (**BP-RP**/**BP-G**/**G-RP**). **Gmag**/**BPmag**/**RPmag**, for example, have high correlation as they are different measurements of stellar brightness in various bands, suggesting that as the brightness in one band increases, it similarly increases in another.

This suggests that these groups provide redundant information and can lead to *multicollinearity* in some models, making them ideal candidates for **PCA**, which can be used to create a single composite feature capturing most of the information from these groups. Moreover, due to their same scales, units, and characteristics, it's unnecessary to scale these groups before applying **PCA**.

```python
from sklearn.decomposition import PCA

def pca_grouping_train(X_train, pca_features={}): # PCA for .fit_transform() on training data
    # Drop original features and concatenate the PCA components
    X_train_new = [X_train.drop(columns=sum(pca_features.values(), []))] # Drop original features
    pca_dict = {} # Store the PCA objects for .transform() on testing data

    # Applying PCA to reduce dimensionality while trying to retain most of the variance in the data
    for group_name, feature_names in pca_features.items():
        pca = PCA(n_components=1) # Reduce to 1 component for simplicity
        pca_feature_train = pca.fit_transform(X_train[feature_names])
        pca_feature_train = pd.DataFrame(pca_feature_train, columns=[group_name], index=X_train.index)

        pca_dict[group_name] = pca # Store the PCA object for .transform() on testing data
        X_train_new.append(pca_feature_train)
        print(f'The new {group_name} feature explains {pca.explained_variance_ratio_[0]*100:.2f}% of the variance')
    return pca_dict, pd.concat(X_train_new, axis=1)


def pca_grouping_test(X_test, pca_dict, pca_features={}): # PCA for .transform() on testing data
    X_test_new = [X_test.drop(columns=sum(pca_features.values(), []))]
    for group_name, feature_names in pca_features.items():
        pca_feature_test = pca_dict[group_name].transform(X_test[feature_names])
        pca_feature_test = pd.DataFrame(pca_feature_test, columns=[group_name], index=X_test.index)
        X_test_new.append(pca_feature_test)
    return pd.concat(X_test_new, axis=1)
```

```python
pca_features = {
    'pca_mag': ['Gmag', 'BPmag', 'RPmag'], # Grouping magnitudes
    'pca_color': ['BP-RP', 'BP-G', 'G-RP'], # Grouping colors
    'pca_error': ['e_Gmag', 'e_BPmag', 'e_RPmag'] # Grouping errors
}

pca_dict, X_train = pca_grouping_train(X_train, pca_features)
X_test = pca_grouping_test(X_test, pca_dict, pca_features)
print(f'The dataset now has {X_train.shape[1]} features:\n{X_train.columns}')
```

```
The new pca_mag feature explains 99.66% of the variance
The new pca_color feature explains 99.46% of the variance
The new pca_error feature explains 95.67% of the variance
The dataset now has 16 features:
Index(['RA_ICRS', 'DE_ICRS', 'Plx', 'PM', 'pmRA', 'pmDE', 'Teff', 'Dist',
       'Rad', 'Lum-Flame', 'Mass-Flame', 'Age-Flame', 'z-Flame', 'pca_mag',
       'pca_color', 'pca_error'],
```

The **PCA** reduced **9 features** in **3 groups** above into **3 principal components** (**pca_mag**/**pca_color**/**pca_error**) while retaining more than **95%** of original variance for each group. This reduction suggests a significant portion of features' variability can be explained with fewer dimensions. The analysis also resulted in **16 main features** for the entire dataset now, making it more manageable and can potentially improve the efficiency of ML algorithms by focusing on the most informative dimensions.

6. Feature Scaling

This step is crucial for optimizing the performance of ML models, particularly those sensitive to feature magnitude like **SVM** or **KNN**. Given the diverse range of values across different features, it is essential to scale the data to ensure that no particular feature dominates the model due to its scale:

- **RobustScaler** will be applied for features with *outliers*, as it uses the *median* and *quartile range* for scaling, making it less sensitive to *outliers*.
- **StandardScaler** for other features to normalize their distributions around zero mean and unit variance.

```python
from sklearn.preprocessing import RobustScaler, StandardScaler
robust_scaler = RobustScaler() # for features containing outliers
standard_scaler = StandardScaler() # for other numeric features

robust_cols = outliers_summary[
    ~outliers_summary.index.isin(sum(pca_features.values(), [])) & # Exclude 3 feature groups used for PCA as they are grouped above
    (outliers_summary.outliers_count_after > 0) # Exclude columns without outliers
].index.tolist() + ['pca_error'] # Include the new pca_error feature
standard_cols = X_train.select_dtypes(include=[np.number]).columns.difference(robust_cols)

X_train[robust_cols] = robust_scaler.fit_transform(X_train[robust_cols])
X_train[standard_cols] = standard_scaler.fit_transform(X_train[standard_cols])
X_test[robust_cols] = robust_scaler.transform(X_test[robust_cols])
X_test[standard_cols] = standard_scaler.transform(X_test[standard_cols])
stats_summary(X_train).round(2)
```

	count	mean	std	min	25%	50%	75%	max	variance	iqr_size	skewness	kurtosis	nulls_count	outliers_count	nulls_percent	outliers_percent
RA_ICRS	118830.0	-0.00	1.00	-1.07	-0.76	-0.41	1.08	2.03	1.00	1.85	0.91	-0.82	0	0	0.0	0.00
DE_ICRS	118830.0	-0.23	0.86	-2.23	-0.55	0.00	0.45	0.69	0.74	1.00	-1.10	0.14	0	9140	0.0	7.69
Plx	118830.0	0.30	0.78	-0.49	-0.31	0.00	0.69	2.35	0.61	1.00	1.22	0.65	0	6672	0.0	5.61
PM	118830.0	0.31	0.86	-0.54	-0.32	0.00	0.68	2.62	0.73	1.00	1.35	0.99	0	7879	0.0	6.63
pmRA	118830.0	0.01	1.00	-1.98	-0.54	0.00	0.46	2.39	1.01	1.00	0.32	0.43	0	7566	0.0	6.37
pmDE	118830.0	-0.26	0.91	-2.50	-0.69	0.00	0.31	1.18	0.83	1.00	-0.90	0.31	0	7555	0.0	6.36
Teff	118830.0	0.04	0.76	-0.85	-0.63	-0.00	0.37	2.01	0.57	1.00	0.98	0.48	0	6235	0.0	5.25
Dist	118830.0	-0.00	1.00	-1.16	-0.88	-0.30	0.76	2.12	1.00	1.63	0.66	-0.78	0	0	0.0	0.00
Rad	118830.0	0.00	1.00	-1.17	-0.77	-0.30	0.53	2.43	1.00	1.30	1.00	0.10	0	0	0.0	0.00
Lum-Flame	118830.0	0.65	1.62	-0.40	-0.28	0.00	0.72	6.04	2.61	1.00	2.32	4.60	0	13628	0.0	11.47
Mass-Flame	118830.0	0.17	0.80	-0.77	-0.46	0.00	0.54	2.26	0.65	1.00	1.09	0.60	0	7039	0.0	5.92
Age-Flame	118830.0	-0.00	1.00	-0.99	-0.83	-0.45	0.83	2.06	1.00	1.65	0.81	-0.78	0	0	0.0	0.00
z-Flame	118830.0	0.00	1.00	-1.52	-0.70	-0.16	0.53	2.24	1.00	1.23	0.63	-0.23	0	0	0.0	0.00
pca_mag	118830.0	0.00	1.00	-1.67	-0.82	-0.00	0.77	1.78	1.00	1.60	0.04	-1.08	0	0	0.0	0.00
pca_color	118830.0	0.00	1.00	-1.48	-0.78	-0.18	0.67	2.10	1.00	1.45	0.52	-0.62	0	0	0.0	0.00
pca_error	118830.0	0.94	2.12	-0.26	-0.19	0.00	0.81	8.07	4.49	1.00	2.31	4.32	0	17210	0.0	14.48

The data has been scaled with most features now having a **mean of ~0 and a std of ~1**, bringing them closer to a similar scale which benefits many ML algorithms.

II. How to Approach the Problem

1. Cross-validation on Base Classifiers for Initial Testing

I start by using various simple to complex models with their **default parameters** from **sklearn** to establish a **baseline** understanding of what works well for the dataset. Each model has strengths in handling different types of data and classification boundaries. By performing this **Initial Testing** with *cross-validation*, I can assess each model's generalization ability and identify issues like *overfitting* or *underfitting*.

The **StratifiedKFold** (default **sklearn** cross-validator) will be applied here, with **5 folds** to ensure each fold is a good representative of the whole, maintaining the same proportion of samples in each class, particularly important in *imbalanced* datasets. This method helps in assessing the model robustly by understanding how the model performs across different subsets of the dataset.

F1-score will be chosen as the **main metric** for this step and for later *cross-validations* and *hyperparameter tuning* as well. Normally, for classification, the most common measure is **Accuracy**. However, when the data is *imbalanced*, **Accuracy** is not a good measure, it should be replaced by **Precision** and **Recall**. For example:

$$Accuracy = \frac{True\ Positive + True\ Negative}{Total\ Samples} = \frac{55 + 850}{1000}$$
$$= 90.5\%\ (Dominated\ by\ 850)$$

Precision and **Recall** can help evaluate the rate of correct "positive" prediction out of the total number of cases predicted to be "positive" and the rate of correct "positive" prediction in reality. However, it is difficult to say whether **Precision** or **Recall** is better, so **F1-score** will be used as the *harmonic average* representing both **Precision** and **Recall**. This ensures both the presence and absence of class are predicted accurately.

During this **Initial Testing**, models will be evaluated based on the *fitting time* and the **mean F1-scores** for both **training & validation set**. Signs of *overfitting* (where **train F1** >> **validation F1**) and *underfitting* (where **both F1** are low) are also taken into considerations. This step helps in shortlisting the most promising models based on their ability to generalize well to new data.

2. Hyperparameter Tuning of Potential Classifiers

This step will find the **best parameter settings** for potential models identified in the **Initial Testing** phase by tuning their hyperparameters to maximize **F1-score**. **HalvingRandomSearchCV** is applied here. It's an efficient approach for parameter tuning that uses *successive halving* to narrow down the search for optimal parameters. This method is more efficient than traditional **GridSearchCV**. StratifiedKFold is also applied for the *cross-validation* in this tuning process to maintain robustness and reliability of the model performance across data subsets.

3. Evaluation for Best Estimator from Each Search

This utilizes a range of metrics to evaluate each model's performance on the **test set**, providing a balanced view of how well models predict across different scenarios. This helps in choosing the **best estimators** from the searches of above potential classifiers. Below is a list of metrics will be considered:

- **Accuracy**: Measure the overall correctness of models across all classes. While straightforward, it can be misleading in *imbalanced* datasets.
- **Precision**: Indicate the correctness achieved in *positive* prediction (i.e., how many *predicted positives* are *truly positive*).
- **Recall**: Measure the proportion of *actual positives* that were accurately classified.
- **F1-Score**: Harmonic means of **Precision** & **Recall**, capturing the balance between them, particularly useful when classes are *imbalanced*.
- **Confusion Matrix**: Provide a visual understanding of prediction results beyond simple **Accuracy**. The number of *right* & *wrong* predictions is reported using *count values* and broken down by class.
- **ROC-AUC**: Reflect models' ability to discriminate *positive* and *negative* classes across threshold settings. Higher values indicate better discriminatory capabilities.

4. Further Improvement with Soft Voting

This step combines the strengths of individually optimized models to create a more robust and accurate classifier by using a **Soft Voting** mechanism averaging their probabilistic predictions. A **VotingClassifier** will be configured so that each classifier's output (probabilities) is taken into account with a weight that reflects its relative

confidence in each prediction. **HalvingRandomSearchCV** will be conducted on the different weight configurations assigned to each classifier in the **Soft Voting** setup to find the best mixture that optimizes the **F1-score**. Finally, the same thorough evaluation process in **part 3** will be applied to this ensemble model, ensuring it meets or exceeds the performance of individual classifiers. This method typically provides a performance boost as it leverages the predictive power of various classifiers.

The **best model** will finally be used to make predictions on the **unknown data**, using the **same preprocessing** steps as were used for the **training set** to ensure consistency and reliability in predictions.

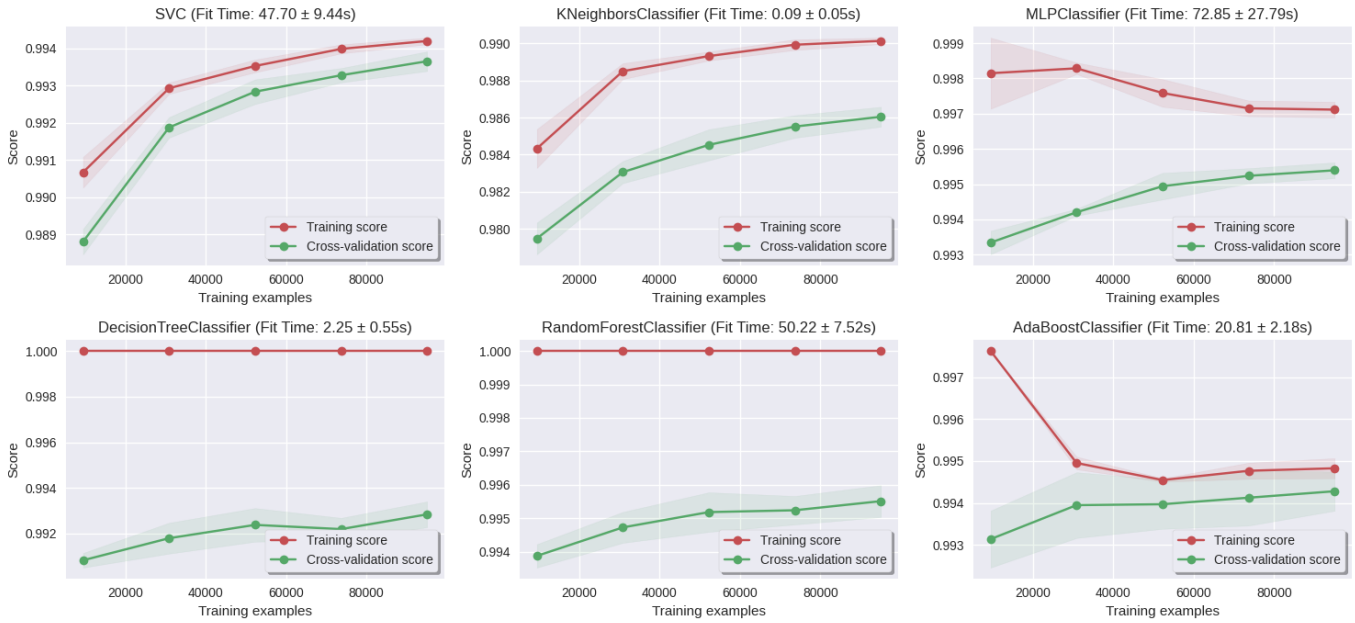
III. Techniques Used, Summary of Results, and Parameter Settings

1. Cross-validation on Base Classifiers for Initial Testing

```python
def plot_learning_curve(clf, X, y, ax=None, ylim=None, cv=None, train_sizes=np.linspace(.1, 1.0, 5), scoring='f1'):
    if ax is None: _, ax = plt.subplots(1, 1, figsize=(15, 5))
    if ylim is not None: ax.set_ylim(*ylim)
    train_sizes, train_scores, test_scores, fit_times, _ = learning_curve(
        clf, X, y, cv=cv, train_sizes=train_sizes, scoring=scoring, return_times=True, verbose=2, n_jobs=-1
    )
    # Mean and standard deviation of training and test scores
    mean_train_scores, std_train_scores = np.mean(train_scores, axis=1), np.std(train_scores, axis=1)
    mean_test_scores , std_test_scores  = np.mean(test_scores, axis=1) , np.std(test_scores, axis=1)
    mean_fit_times   , std_fit_times    = np.mean(fit_times, axis=1)   , np.std(fit_times, axis=1)

    ax.grid(visible=True) # Plot learning curves
    ax.fill_between(train_sizes, mean_train_scores - std_train_scores, mean_train_scores + std_train_scores, alpha=0.1, color='r')
    ax.fill_between(train_sizes, mean_test_scores - std_test_scores, mean_test_scores + std_test_scores, alpha=0.1, color='g')
    ax.plot(train_sizes, mean_train_scores, 'o-', color='r', label='Training score')
    ax.plot(train_sizes, mean_test_scores, 'o-', color='g', label='Cross-validation score')
    ax.set_title(f'{clf.__class__.__name__} (Fit Time: {mean_fit_times[-1]:.2f} ± {std_fit_times[-1]:.2f}s)')

    ax.legend(loc='lower right', frameon=True, shadow=True)
    ax.set_xlabel('Training examples')
    ax.set_ylabel('Score')
    return {
        'model_name': clf.__class__.__name__, # Last element corresponds to the full dataset cross-validation
        'mean_train_score': mean_train_scores[-1], 'mean_test_score': mean_test_scores[-1], 'mean_fit_time': mean_fit_times[-1],
        'std_train_score' : std_train_scores[-1] , 'std_test_score' : std_test_scores[-1] , 'std_fit_time' : std_fit_times[-1]
    }
```

```python
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.model_selection import learning_curve
from tqdm.notebook import tqdm

# Initializing classifiers with default parameters
classifiers = [
    SVC(), KNeighborsClassifier(n_jobs=-1), MLPClassifier(),
    DecisionTreeClassifier(), RandomForestClassifier(n_jobs=-1), AdaBoostClassifier()
]
fig, axes = plt.subplots(2, 3, figsize=(15, 7))
axes = axes.ravel()
cv_results = []

for idx, clf in enumerate(tqdm(classifiers)):
    f1_result = plot_learning_curve(clf, X_train, y_train, ax=axes[idx], cv=5)
    cv_results.append(f1_result)
plt.tight_layout()
plt.show()
```

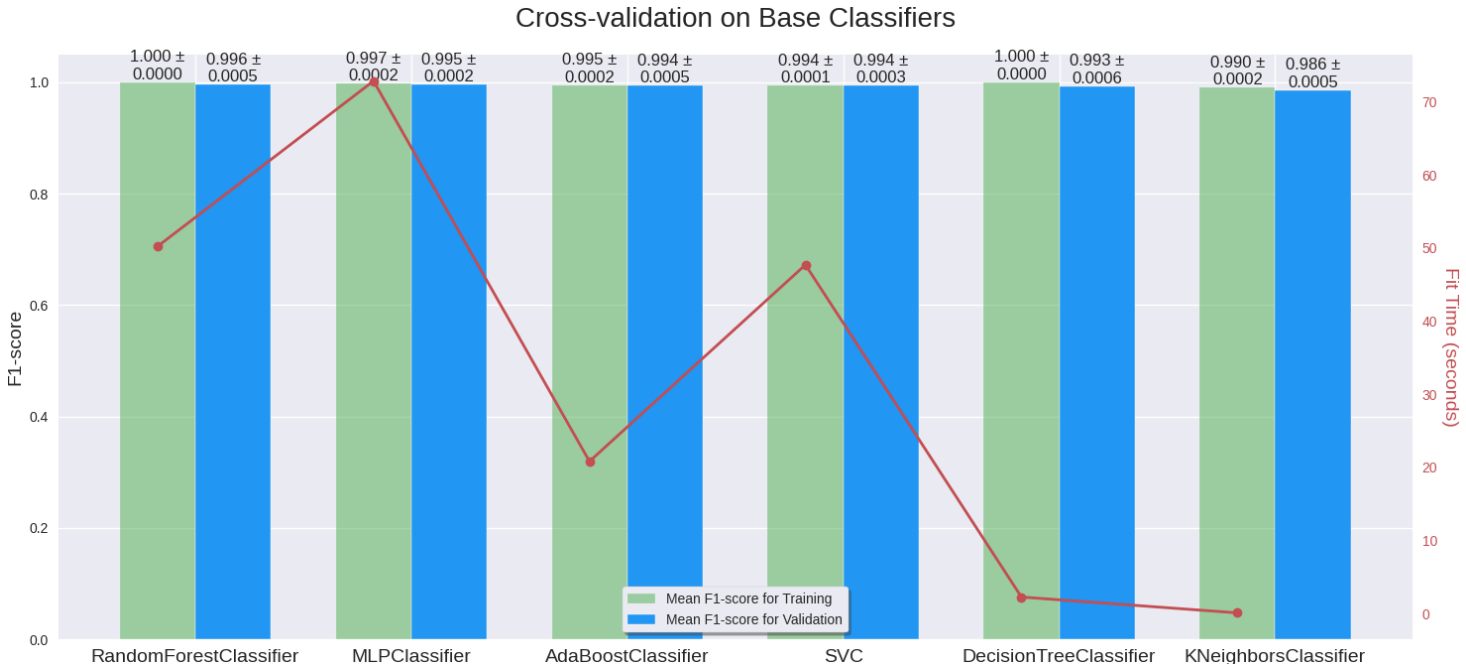
```
  0%|          | 0/6 [00:00<?, ?it/s]

[learning_curve] Training set sizes: [ 9506 30895 52285 73674 95064]
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done  13 out of  25 | elapsed:  7.5min remaining:  6.9min
```



	model_name	mean_train_score	mean_test_score	mean_fit_time	std_train_score	std_test_score	std_fit_time
0	RandomForestClassifier	1.000000	0.995512	50.224332	0.000000	0.000476	7.519947
1	MLPClassifier	0.997115	0.995393	72.850752	0.000222	0.000224	27.793389
2	AdaBoostClassifier	0.994827	0.994279	20.810329	0.000239	0.000466	2.183351
3	SVC	0.994195	0.993652	47.702046	0.000089	0.000263	9.436550
4	DecisionTreeClassifier	1.000000	0.992841	2.253132	0.000000	0.000563	0.545505
5	KNeighborsClassifier	0.990123	0.986040	0.085705	0.000170	0.000531	0.052968

Here, I performed an initial evaluation of **6 classifiers** by plotting their *learning curve* using their default settings from **sklearn** to visualize the models' performance as the training size increases. These curves can also make it easy to identify the signs of *overfitting* or *underfitting*. The "**learning_curve**" function from **sklearn** can even utilize **StratifiedKFold** *cross-validation* to ensure each fold had same class distribution and models' performance is consistent This helps to establish a **baseline** performance, avoiding *overfitting* and ensuring the model generalizes well to new data. I then used this information to identify the most promising models for further optimization & tuning.

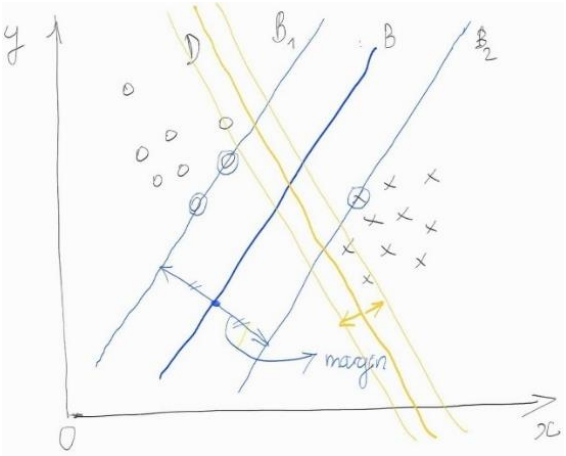
```python
def plot_cv_results(cv_results, title='Cross-validation', bar_width=0.35, fontsize=14, figsize=(15, 7)):
    plt.figure(figsize=figsize)
    index = np.arange(len(cv_results['model_name']))
    train_bars = plt.bar(index, cv_results['mean_train_score'], bar_width, label='Mean F1-score for Training', color='#4CAF50', alpha=0.5)
    test_bars = plt.bar(index + bar_width, cv_results['mean_test_score'], bar_width, label='Mean F1-score for Validation', color='#2196F3')
    plt.title(title, fontsize=fontsize+6, pad=20)
    plt.ylabel('F1-score', fontsize=fontsize)
    plt.xticks(index + bar_width / 2, cv_results['model_name'], ha='center', fontsize=fontsize)
    plt.legend(loc='lower center', frameon=True, shadow=True)

    # Adding the data labels on the bars
    for idx, bars in enumerate((train_bars, test_bars)):
        stds = cv_results['std_test_score'] if idx else cv_results['std_train_score']
        for bar, std in zip(bars, stds):
            f1_mean = bar.get_height()
            plt.text(bar.get_x() + bar.get_width()/2, f1_mean, f'{f1_mean:.3f} ±\n{std:.4f}', fontsize=fontsize-2, va='bottom', ha='center')

    ax2 = plt.gca().twinx() # Adding secondary axis for fit time
    ax2.plot(cv_results['model_name'], cv_results['mean_fit_time'], color='r', marker='o', label='Fit Time Mean (s)', linewidth=2)
    ax2.set_ylabel('Fit Time (seconds)', color='r', fontsize=fontsize, rotation=270, labelpad=15)
    ax2.tick_params(axis='y', labelcolor='r')
    ax2.yaxis.set_label_position('right')
    ax2.grid(False)
    plt.tight_layout()
    plt.show()
```



a) Support Vector Machine (SVC)

The first chosen model is **SVM**, a supervised ML algorithm that works by finding the optimal **hyperplane** maximizing the margin between different classes in a dataset. Mathematically, given training data (x_i, y_i) where $x_i \in \mathbb{R}^n$ and $y_i \in \{-1, 1\}$, SVM aims to solve the optimization problem: $\min_{w,b} \frac{1}{2}\|w\|^2 \; subject\ to\ y_i(w.x_i + b) \geq 1 \; \forall i.$

The solution to this problem involves finding *w* and *b* that maximize the distance (margin) between the **hyperplane** & each class' closest data points (**support vectors**). For non-linearly separable data, SVMs can employ *kernel tricks* (e.g., polynomial, RBF) to transform the data into higher-dimension where it becomes linearly separable and uses regularization to avoid *overfitting*. The dual formulation of SVM, which uses Lagrange multipliers, allows for efficient computation and flexibility in incorporating different kernel functions.



Its *learning curve* indicates a consistent improvement in training & cross-validation scores with increasing data but took considerable fitting time (47.70s) compared to some other models. The F1-score on the train and validation set were 0.994 and 0.993625, respectively. This indicates strong performance but just small potential of *overfitting*. This could be due to the complexity of finding the optimal hyperplane in high-dimensional space. However, the longer training time might be a consideration.

b) K-Nearest Neighbors (KNeighborsClassifier)

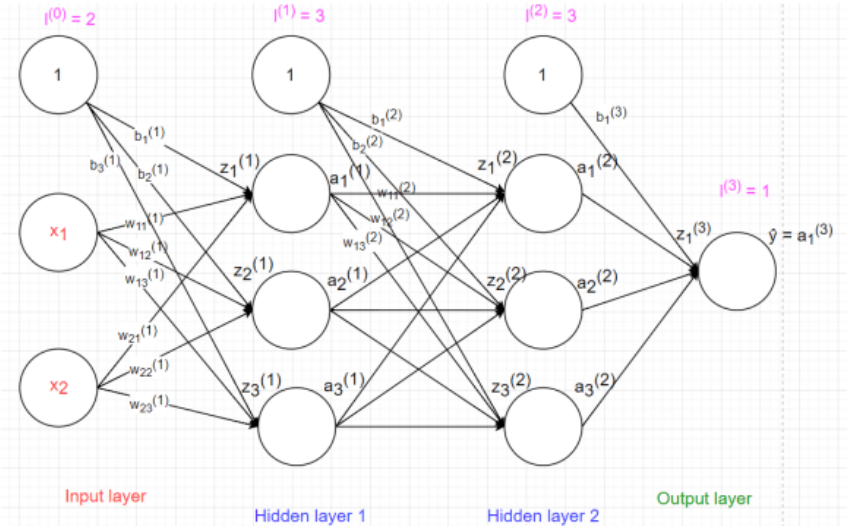
KNN is a straightforward method that classifies new data points based on their similarity to known data. For example, when plotting all data on a graph, each data point has a location defined by its features. When a new data point arrives, KNN simply looks at the **'k'** closest data points (its neighbors) in this space. The **closeness** is calculated by a similarity metric, often Euclidean distance: $\sqrt{\sum(\mathrm{x}_i - y_i)^2}$, where *x* and *y* are corresponding feature values of the 2 data points. The new point is then assigned the majority class (classification) or average value (regression) of its **k-neighbors**. Essentially, KNN assumes similar things exist in close proximity, making it a lazy learner that memorizes the training data instead of building a complex model.

The KNN showed good training and validation scores with **shortest** training time (0.09 seconds), indicating its efficency with quick computations. However, the model's generalization capability was weak compared to others, suggesting the model is too simple to capture the underlying patterns effectively, so it might not be a robust model for more complex data patterns in this dataset.

c) Multi-layer Perceptron Classifier (MLPClassifier)

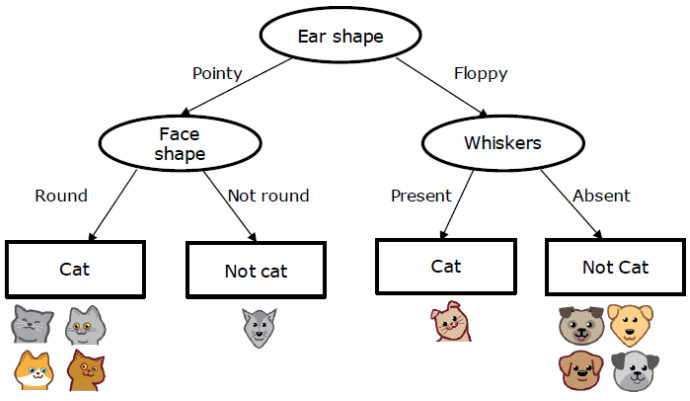
This is a type of artificial neural network that learns complex patterns or non-linear relationships in data by stacking layers of interconnected **neurons**. It is made up of many layers of perceptrons with at least 3 levels of nodes: Input layer, Hidden layer, and Output layer, all of which are fully linked to the next. Each neuron takes inputs, multiplies them by weights (representing the strength of connections), adds bias, and applies *activation* functions (*sigmoid*, *ReLU,* …) to generate an output. These outputs are then fed to next layer, allowing the network to learn progressively more abstract representations. The weights are updated during training using optimization algorithm like *Gradient Descent* to minimize the loss function (compute the difference between the predictions and actual values). This **backpropagation** process involves calculating

gradients of the loss function with respect to each weight, allowing the network to refine its internal connections and improve its accuracy.



MLP achieved high performance with a validatied F1-score of 0.995393 but it had the highest fitting time (72.85s). The model's complexity allows it to learn complex patterns effectively, resulting in high validation score close to the training score (but this requires careful tuning to avoid *overfitting*). Due to its performance, this model might benefit from later tunings to further optimize the performance. Regularization parameters like **"alpha"** can help mitigate *overfitting*.

d) Decision Tree (DecisionTreeClassifier)

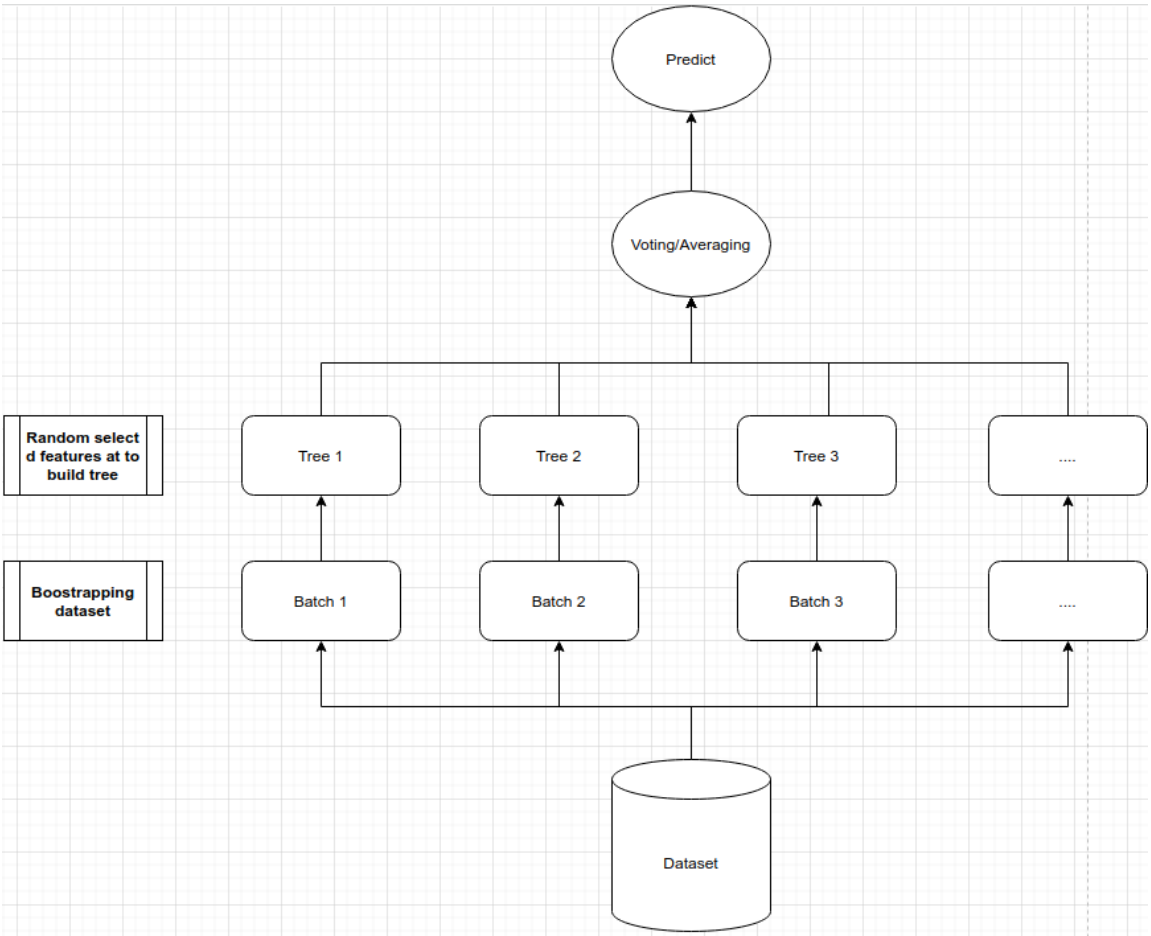


This is a powerful ML algorithm that, much like a flow chart, break down complex datasets into a series of increasingly specific decisions. At each node, the algorithm chooses the best feature (based on metrics like Gini impurity or entropy) to split the data, maximizing homogeneity within resulting branches. The impurity measures, using concepts like probability & **information gain**, quantify the *mixedness* of classes within a node. This process continues **recursively** until a stopping condition is satisfied, like satisfying a predefined depth or a minimum number of samples in each leaf. The final result is a tree structure, with each path from the root to a leaf representing a unique decision rule, ultimately classifying/predicting target variable.

Decision Tree showed perfect training score of 1.0 but slightly lower mean validation score of 0.992841, indicating potential *overfitting*. This suggests the model memorizes the training data rather than learning generalizable patterns. Its fitting time was very low (2.25 seconds), making it computationally efficient.

e) Random Forest (RandomForestClassifier)

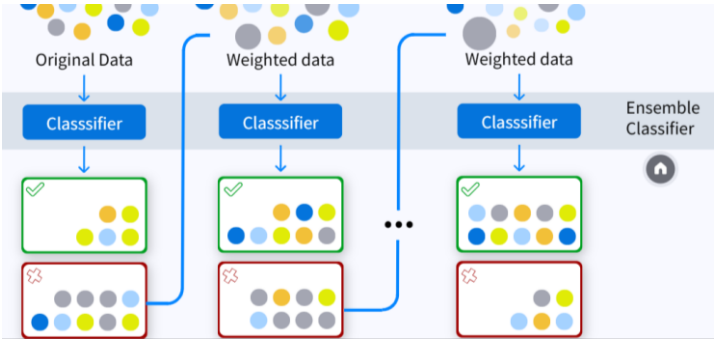
This is a powerful **ensemble learning** technique leveraging the “wisdom of the crowd” by aggregating predictions from many Decision Trees. Each tree is trained on a random subset of features and data, encouraging diversity and reducing *overfitting*. This **bagging** process, known as **bootstrap** aggregating, involves *sampling with replacement*, where each data point has a chance of being selected multiple times within a single tree's training set. **Feature randomness** further enhances the model's robustness by choosing a random subset of features for every split within a tree. By averaging predictions of these diverse trees, it effectively reduces variance, mitigating the impact of noise and outliers. The final prediction is determined by a majority vote (classification) or averaging (regression), allowing the model to handle complex relationships within the data while mitigating the risk of single-tree biases.



Random Forest had the **best performance** with an average validation F1-score of 0.995512, indicating its robustness and strong generalization across different subsets of the data. It had moderate fitting times (50.22s), which is acceptable given its performance, making it a robust choice for further classification.

f) Adaptive Boosting (AdaBoostClassifier)

This is another **ensemble learning** approach combining many weak classifiers (each performing slightly better than random guessing) to form a strong one. It works by fitting series of **weak learners** on repeatedly modified versions of data and adjusting the weights of incorrectly classified instances, focusing more on **hard** samples. The final prediction is a weighted sum of predictions from all weak learners, where the weights reflect the learner's performance. This adaptive weighting scheme allows AdaBoost to achieve remarkable accuracy, particularly on complex datasets.



AdaBoost shows strong performance that is close to MLPClassifier with a mean validation score of 0.994279 but with better training time of 20.81 seconds. It effectively balances accuracy and training time, showing minimal *overfitting*. The model's training score was slightly lower than others, but it showed good generalization in overall.

g) Summary

Overall, most classifiers show a high mean train score close to **1.0**, indicating they fit the training data well. However, the key is to look at the mean validation scores to gauge how well the model generalizes. Models like **Decision Tree** showed signs of *overfitting* (1.0 training score, 0.992841 validation score), whereas **Random Forest** and **AdaBoost** demonstrated better generalization (closer training and validation scores). Regarding the training time, **KNN** has the shortest fit time but sacrifices accuracy, while **MLP** and **SVC** have longer fit times but offer higher accuracy. **Random Forest** and **AdaBoost** offer a good balance between fit time and performance. Balancing performance and computational cost are essential.

The **PCA** from the Preprocessing step was a strategic move. This dimensionality reduction is crucial for enhancing model efficiency and avoiding *overfitting*, especially in this high-dimensional dataset. Here, only **16** principal components were used, significantly reducing the dimensionality from the original feature set. The **baseline** results show the effectiveness of **PCA** in minimizing noise & computational complexity while retaining the most informative aspects needed for accurate classification.

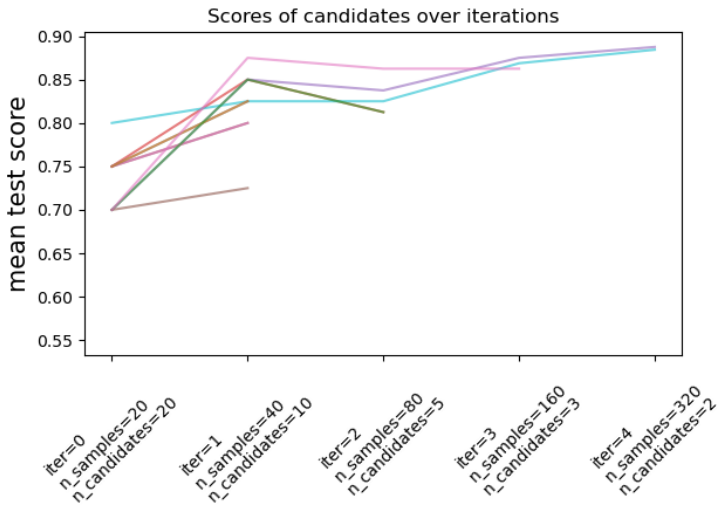
From this **Initial Testing**, **Random Forest**, **MLP**, and **AdaBoost** emerge as the most promising classifiers based on their high **F1-scores** with reasonable training times. These models will be taken for further tuning to ensure optimal performance.

2. Hyperparameter Tuning for Potential Classifiers

In this step, the goal is to find the best hyperparameters and perform relevant training for 3 promising classifiers identified during the Initial Testing: **MLPClassifier**, **RandomForestClassifier** & **AdaBoostClassifier** by using a powerful and resource-efficient search strategy, **Halving Search**.

a) Halving Search

Halving Search is an iterative approach for hyperparameter tuning that begins with a large pool of candidate configurations (settings) and iteratively reduces the number of candidates by halving the least promising ones. The process involves evaluating each candidate on a small subset of data and progressively allocating more resources (e.g., number of iterations, data samples) to the better-performing candidates in subsequent iterations. This approach balances exploration of the hyperparameter space and exploitation of the best configurations found so far. It can be applied in 2 variants: **HalvingGridSearchCV** and **HalvingRandomSearchCV**.



In this step, **HalvingRandomSearchCV** will be applied as it outperforms other search methods like **GridSearchCV**, **RandomSearchCV**, and **HalvingGridSearchCV** by combining the efficiency of random sampling with an iterative process that progressively focuses on the most promising candidates. Unlike the traditional **GridSearchCV**, which exhaustively searches over a fixed set of parameters, and **RandomSearchCV**, which samples randomly across the entire search space, **HalvingRandomSearchCV** starts with a larger set of random configurations and then narrows them down in successive iterations. This method reduces computational costs by allocating resources dynamically, allowing for quicker convergence to optimal solutions. Compared to **HalvingGridSearchCV**, which similarly reduces the search space iteratively but starts with a grid-based approach, **HalvingRandomSearchCV** offers greater flexibility and a higher likelihood of finding better hyperparameter combinations in fewer iterations due to its initial random sampling.

b) Important Settings for HalvingRandomSearchCV

```python
from sklearn.experimental import enable_halving_search_cv
from sklearn.model_selection import HalvingRandomSearchCV

potential_classifiers = {
    'MLPClassifier': MLPClassifier(),
    'RandomForestClassifier': RandomForestClassifier(n_jobs=-1),
    'AdaBoostClassifier':  AdaBoostClassifier(),
}

param_distributions = {
    'MLPClassifier': {
        'hidden_layer_sizes': [(100,), (100, 100)], 'alpha': [0.0001, 0.001],
        'learning_rate_init': [0.001, 0.01, 0.1], 'learning_rate': ['constant', 'adaptive'],
    },
    'RandomForestClassifier': {
        'n_estimators': [100, 200, 300], 'class_weight': ['balanced', None],
        'max_features': ['sqrt', 'log2', None]
    },
    'AdaBoostClassifier': {'n_estimators': [100, 200, 300], 'learning_rate': [0.01, 0.05, 0.1, 0.5, 1]},

}

searchers = {}
for name, clf in tqdm(potential_classifiers.items()):
    searchers[name] = HalvingRandomSearchCV(
        clf, param_distributions[name], resource='n_samples', factor=3, cv=5,
        scoring='f1', return_train_score=True, random_state=42, verbose=3
    ).fit(X_train, y_train)
```

```
  0%|          | 0/3 [00:00<?, ?it/s]

n_iterations: 3
n_required_iterations: 3
n_possible_iterations: 8
min_resources_: 20
max_resources_: 118830
aggressive_elimination: False
factor: 3
----------
iter: 0
n_candidates: 24
n_resources: 20
Fitting 5 folds for each of 24 candidates, totalling 120 fits
[CV 1/5] END alpha=0.0001, hidden_layer_sizes=(100,), learning_rate=constant, learning_rate_init=0.001;, score=(train=1.000, test=0.500) total time=   0.1s
[CV 2/5] END alpha=0.0001, hidden_layer_sizes=(100,), learning_rate=constant, learning_rate_init=0.001;, score=(train=1.000, test=1.000) total time=   0.1s
[CV 3/5] END alpha=0.0001, hidden_layer_sizes=(100,), learning_rate=constant, learning_rate_init=0.001;, score=(train=1.000, test=1.000) total time=   0.1s
[CV 4/5] END alpha=0.0001, hidden_layer_sizes=(100,), learning_rate=constant, learning_rate_init=0.001;, score=(train=1.000, test=1.000) total time=   0.1s
[CV 5/5] END alpha=0.0001, hidden_layer_sizes=(100,), learning_rate=constant, learning_rate_init=0.001;, score=(train=1.000, test=1.000) total time=   0.1s
```

- **Parameter Distributions**: Define the search space for each classifier to explore a wide range of potential hyperparameter values (see below).
- **Resource Parameter**: "**n_samples**" indicates that the resource being halved is the number of training samples, which is progressively increased in each iteration.
- **Factor**: **3**, meaning in each iteration, only the top third of the configurations are retained for the next round and the resource allocation is tripled.
- **CV**: **5-fold StratifiedKFold** with balanced class representation for cross-validation
- **Scoring Metric**: **F1**-score is used to evaluate and rank the configurations.

c) Parameter Distributions

Potential Classifier	Chosen hyperparameters search space
MLPClassifier The configuration chosen is to balance complexity (hidden layers) & regularization (alpha) while experimenting with different learning rates to ensure convergence and avoid *overfitting*.	- **hidden_layer_sizes**: Define the architecture of the neural network with number of neurons in each hidden layer. More layers/neuron can capture more complex patterns. **(100,)** means 1 hidden layer with **100** neurons, while **(100, 100)** means 2 hidden layers, each with **100** neurons. Smaller sizes (e.g., **(100,)**) are simpler, while larger sizes (e.g., **(100, 100)**) can capture more complex patterns. - alpha (0.0001/0.001): L2 regularization term to prevent *overfitting*. - **learning_rate_init (0.001/0.01/0.1):** Initial learning rate for the weight updates. - **learning_rate**: Control the step size during optimization. **constant** means a fixed rate (keep constant), while **adaptive** allows it to change based on performance.
RandomForestClassifier This configuration aims to balance the number of trees & feature selection strategies to ensure robust performance while handling class imbalance.	- **n_estimators (100/200/300)**: Number of trees in the forest. More trees may improve performance but increase computation. - **class_weight**: Adjusts weights to handle class imbalance (**'balanced'**) or not (**None**). - **max_features (sqrt/log2/None)**: Number of features for splitting at each node. **sqrt** for the square root of total features, **log2** for the log base 2, and **None** considers all features. A smaller number can reduce *overfitting*.

AdaBoostClassifier Its setup explores several boosting iterations & learning rates to find optimal balance between performance & *overfitting*.	- **n_estimators (100/200/300)**: Number of weak learners (typically decision stumps). More estimators can improve accuracy but also risk *overfitting*. - **learning_rate (0.01/0.05/0.1/0.5/1)**: Shrink the contribution of each weak learner. Lower values can prevent *overfitting* and improve generalization.

3. Evaluation for Best Estimator from Each Search

The classification reports provide a detailed breakdown of **Precision**, **Recall**, and **F1-scores** for each class (**0** and **1**) for the best-tuned classifiers. All models performed exceptionally well, with slight differences across these metrics, indicating excellent performance, with **RandomForestClassifier** slightly outperforming others:

```python
from sklearn.metrics import classification_report, confusion_matrix, roc_curve, auc
reports, conf_matrices, roc_aucs = {}, {}, {}

for model_name, searcher in searchers.items():
    y_pred = searcher.best_estimator_.predict(X_test)
    y_probs = searcher.best_estimator_.predict_proba(X_test)
    report = classification_report(y_test, y_pred, output_dict=True)
    fpr, tpr, _ = roc_curve(y_test, y_probs[:, 1]) # Use [:, 1] for positive class probabilities

    reports[model_name] = pd.DataFrame(report).T
    conf_matrices[model_name] = confusion_matrix(y_test, y_pred)
    roc_aucs[model_name] = (fpr, tpr, auc(fpr, tpr))
```

```python
html_str, width = '', 100 / len(reports)
for model_name, report_df in reports.items():
    html_str += f"""
    <div style='width:{width}%; float: left; overflow-x: auto;'>
        <center>
            Classification Report for <b>{model_name}</b><br>
            Best settings: <b>{searchers[model_name].best_params_}</b>
        </center>
        {report_df.to_html()}
    </div>"""
display_html(html_str, raw=True)
```

Classification Report for **MLPClassifier**
Best settings: **{'learning_rate_init': 0.01, 'learning_rate': 'constant', 'hidden_layer_sizes': (100,), 'alpha': 0.0001}**

	precision	recall	f1-score	support
0	0.996622	0.994756	0.995688	16018.000000
1	0.993878	0.996056	0.994965	13690.000000
accuracy	0.995355	0.995355	0.995355	0.995355
macro avg	0.995250	0.995406	0.995327	29708.000000
weighted avg	0.995358	0.995355	0.995355	29708.000000

Classification Report for **RandomForestClassifier**
Best settings: **{'n_estimators': 100, 'max_features': 'sqrt', 'class_weight': 'balanced'}**

	precision	recall	f1-score	support
0	0.996564	0.995942	0.996253	16018.000000
1	0.995255	0.995982	0.995619	13690.000000
accuracy	0.995961	0.995961	0.995961	0.995961
macro avg	0.995910	0.995962	0.995936	29708.000000
weighted avg	0.995961	0.995961	0.995961	29708.000000

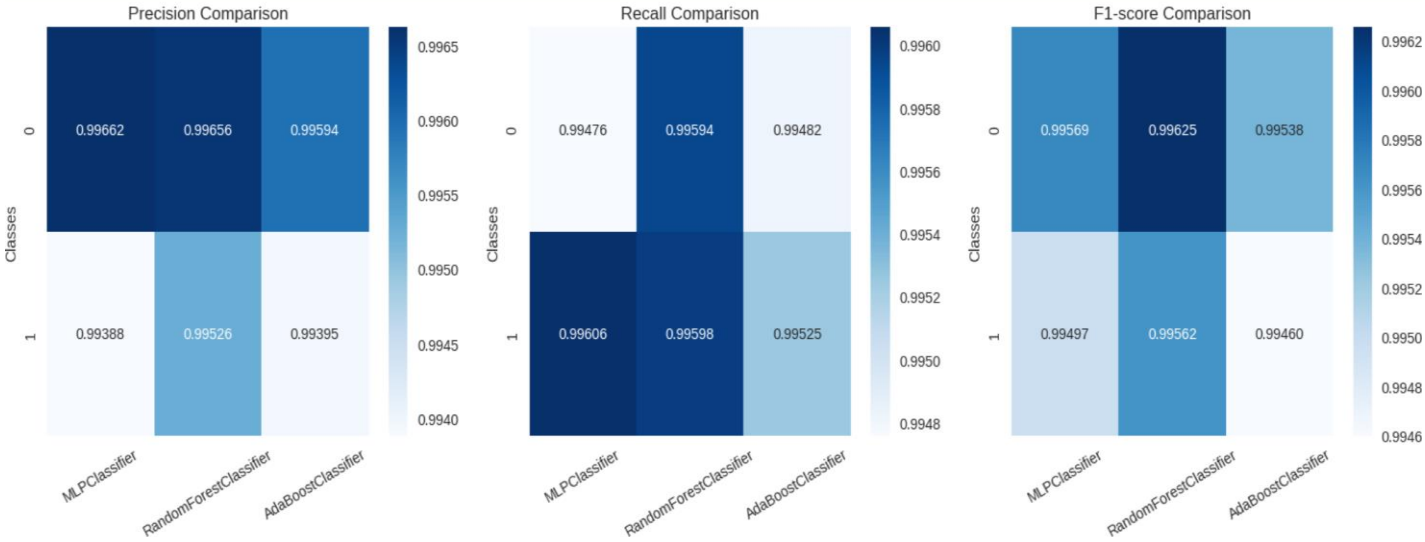
Classification Report for **AdaBoostClassifier**
Best settings: **{'n_estimators': 100, 'learning_rate': 1**

	precision	recall	f1-score	support
0	0.995938	0.994818	0.995378	16018.000000
1	0.993945	0.995252	0.994598	13690.000000
accuracy	0.995018	0.995018	0.995018	0.995018
macro avg	0.994941	0.995035	0.994988	29708.000000
weighted avg	0.995019	0.995018	0.995018	29708.000000

Best Estimator	Performance
MLPClassifier {'learning_rate_init': **0.01**, 'learning_rate': '**constant**', 'hidden_layer_sizes': **(100,)**, 'alpha': **0.0001**}	- **Precision**, **Recall**, and **F1-score** are high across both classes, indicating the model's effectiveness in distinguishing classes. - Slightly lower performance for **class 1** compared to **class 0**, indicating potential class imbalance impact with slightly biased towards **class 0**.

RandomForestClassifier {'n_estimators': **100**, 'max_features': **'sqrt'**, 'class_weight': **'balanced'**}	- This performance can be attributed to its class weight adjustments and its ensemble nature, which combines multiple decision trees to improve robustness and accuracy. - The **balanced class_weight** parameter helps handle any class imbalance, reflected in the high and balanced scores across metrics for both classes, showing a robust generalization with the overall accuracy at **0.996**.
AdaBoostClassifier {'n_estimators': **100**, 'learning_rate': **1**}	- High metrics across both classes, with overall accuracy at **0.995** indicates good generalization. - Similar to **MLP**, slightly lower performance for **class 1**, suggesting minor class imbalance, but overall performance is strong.

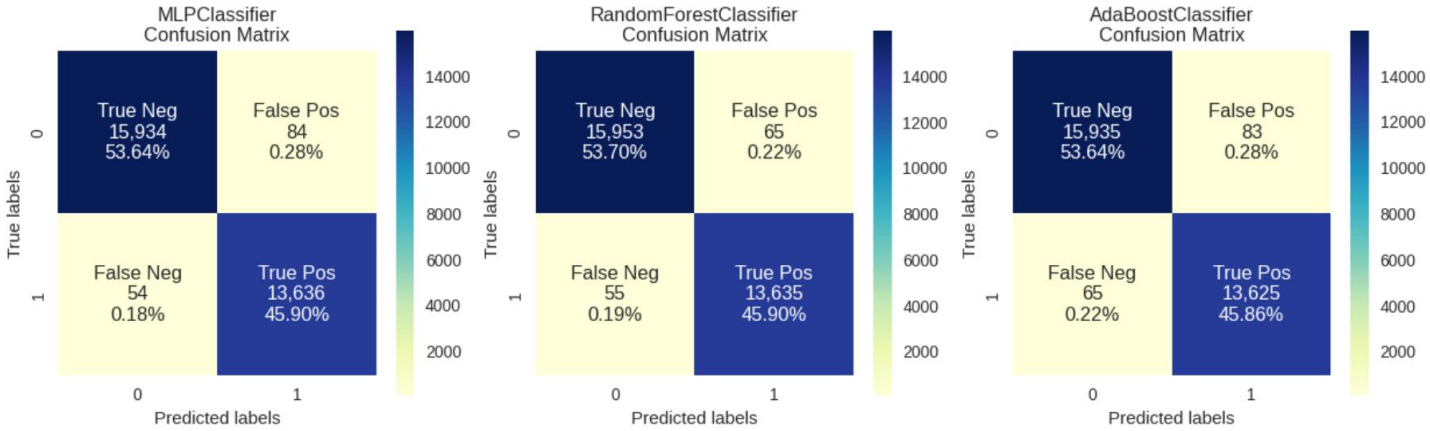
```
metrics = ['precision', 'recall', 'f1-score']
overall_scores = {metric: pd.DataFrame() for metric in metrics}

for metric in metrics:
    for model_name, report_df in reports.items():
        overall_scores[metric][model_name] = report_df[metric]

fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(18, 5))
for ax, metric in zip(axes, metrics):
    sns.heatmap(overall_scores[metric].iloc[:-3, :], annot=True, fmt='.5f', cmap='Blues', ax=ax)
    ax.set_title(f'{metric.capitalize()} Comparison')
    ax.set_ylabel('Classes')
    for label in ax.get_xticklabels(): label.set_rotation(30)
```



- **Precision**: All classifiers show high Precisions (**>0.993**), indicating low **FP** rates. **RandomForest** and **AdaBoost** have marginally higher precision for **Class 0** compared to **MLP**. For **class 1**, **RandomForest** has the **highest Precision**.
- **Recall**: Similar trends are observed in recall with very high scores across classifiers for both classes (**>0.995**), indicating few **FN**. Likewise, **Random Forest** shows the best balance.

- **F1-Score**: As a result, **RandomForestClassifier** maintains **highest F1-scores** across both classes. The slight differences in F1-scores across models highlight the balance each model achieves between precision and recall.
- **Class Balance**: **MLPClassifier** and **AdaBoostClassifier** demonstrate slight sensitivity to class imbalance, reflecting in lower performance for **class 1**. **RandomForestClassifier** consistently shows balanced performance, which indicates robustness to class imbalance, making it a robust choice.

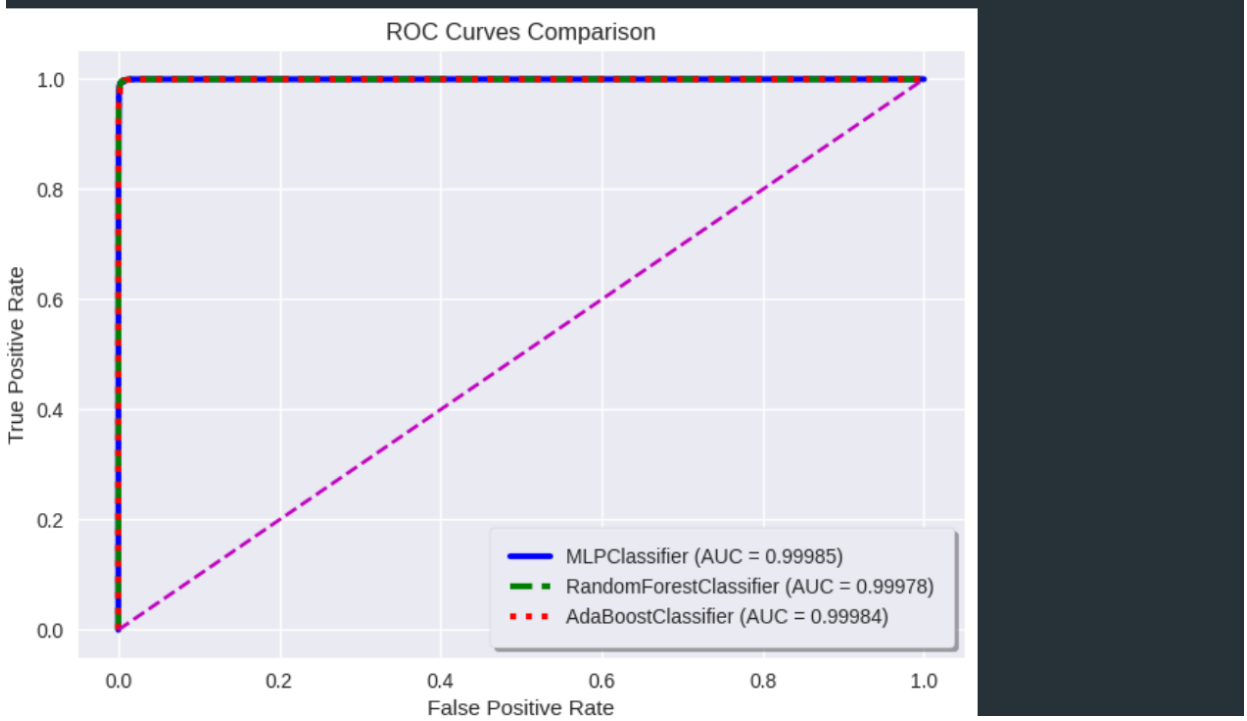
```python
def plot_cf_matrix(cf_matrix, ax=None):
    labels = np.asarray([f'{name}\n{count:,}\n{percent:.2%}' for name, count, percent in zip(
        ['True Neg', 'False Pos', 'False Neg', 'True Pos'], # Group names
        cf_matrix.flatten(), # Group counts
        cf_matrix.flatten() / np.sum(cf_matrix) # Group percentages
    )]).reshape(2, 2)
    sns.heatmap(cf_matrix, fmt='', annot=labels, cmap='YlGnBu', square=True, annot_kws={'size': 12}, ax=ax)

fig, axes = plt.subplots(1, len(searchers), figsize=(5 * len(searchers), 4))
if len(searchers) == 1: axes = [axes] # Make it iterable
for ax, (name, matrix) in zip(axes, conf_matrices.items()):
    plot_cf_matrix(matrix, ax)
    ax.set_title(f'{name}\nConfusion Matrix')
    ax.set_xlabel('Predicted labels')
    ax.set_ylabel('True labels')
plt.show()
```



- **True Positives (TP)** and **True Negatives (TN)**: All classifiers correctly predict a very high number of instances for both classes with high **TN** and **TP** rates, indicate strong predictive performance.
- **False Positives (FP)** and **False Negatives (FN)**: All classifiers have very low **FP** and **FN** rates with the lowest belongs to **RandomForestClassifier**, indicating the best overall performance. The slightly higher **FN** in the **MLPClassifier** suggests it may occasionally miss the minority class (**Class 1**), likely due to overfitting.

Overall, **RandomForestClassifier**'s confusion matrix confirms it is the most robust model, with minimal misclassifications. **MLPClassifier** and **AdaBoostClassifier** show slightly higher misclassifications in **FN** for **Class 1**.

```python
colors = ['b', 'g', 'r', 'c', 'm', 'y']
linestyles = ['-', '--', ':', '-.', '-', '--', ':', '-.']

for (name, (fpr, tpr, score)), color, linestyle in zip(roc_aucs.items(), colors, linestyles):
    plt.plot(fpr, tpr, label=f'{name} (AUC = {score:.5f})', color=color, linestyle=linestyle, lw=3)

plt.plot([0, 1], [0, 1], 'm--')  # Dashed diagonal
plt.title('ROC Curves Comparison')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc='lower right', frameon=True, shadow=True, borderpad=1)
plt.show()
```



The **ROC curve** plot compares the **AUC** (Area Under the Curve) for each classifier. All of them show near-perfect **ROC curves**, with **AUC** values close to the top-left corner (**1.0**), confirming high **TP** rates and low **FP** rates. This indicates they generalize well to unseen data, excellent distinguishing power between classes. **MLPClassifier** has the highest AUC (**0.99985**), closely followed by **AdaBoostClassifier** (**0.99984**) and **RandomForestClassifier** (**0.99978**). These values reflect the high-quality performance of all models without too much difference between them.

4. Further Improvement with Soft Voting

This step aims to improve performance by combining the complementary strengths of individual best-tuned classifiers. The **VotingClassifier** was set up with the optimized estimators from the previous steps. The weights for each classifier during voting were tuned to ensure the optimal contribution of each model to the final prediction, focusing on optimizing the f1-score.

```python
from sklearn.ensemble import VotingClassifier

# Setup Voting Classifier with optimized estimators
optimized_estimators = [(name, searcher.best_estimator_) for name, searcher in searchers.items()]
soft_voting_clf = VotingClassifier(estimators=optimized_estimators, voting='soft', n_jobs=-1)

# Based on the performance of each best estimator, the weights are set to prioritize the best performing models
param_grid = {'weights': [[1, 2, 1], [2, 2, 1], [1, 3, 1], [2, 3, 1], [3, 3, 1]]}

voting_searcher = HalvingRandomSearchCV(
    soft_voting_clf, param_grid, resource='n_samples', factor=3, cv=5,
    scoring='f1', return_train_score=True, random_state=42, verbose=3
).fit(X_train, y_train)
best_voting_clf = voting_searcher.best_estimator_
```

```
n_iterations: 2
n_required_iterations: 2
n_possible_iterations: 8
min_resources_: 20
max_resources_: 118830
aggressive_elimination: False
factor: 3
----------
iter: 0
n_candidates: 5
n_resources: 20
Fitting 5 folds for each of 5 candidates, totalling 25 fits
[CV 1/5] END weights=[1, 2, 1];, score=(train=1.000, test=0.500) total time=   0.4s
[CV 2/5] END weights=[1, 2, 1];, score=(train=1.000, test=1.000) total time=   0.6s
[CV 3/5] END weights=[1, 2, 1];, score=(train=1.000, test=0.000) total time=   0.4s
[CV 4/5] END weights=[1, 2, 1];, score=(train=1.000, test=1.000) total time=   0.4s
[CV 5/5] END weights=[1, 2, 1];, score=(train=1.000, test=0.800) total time=   0.4s
```

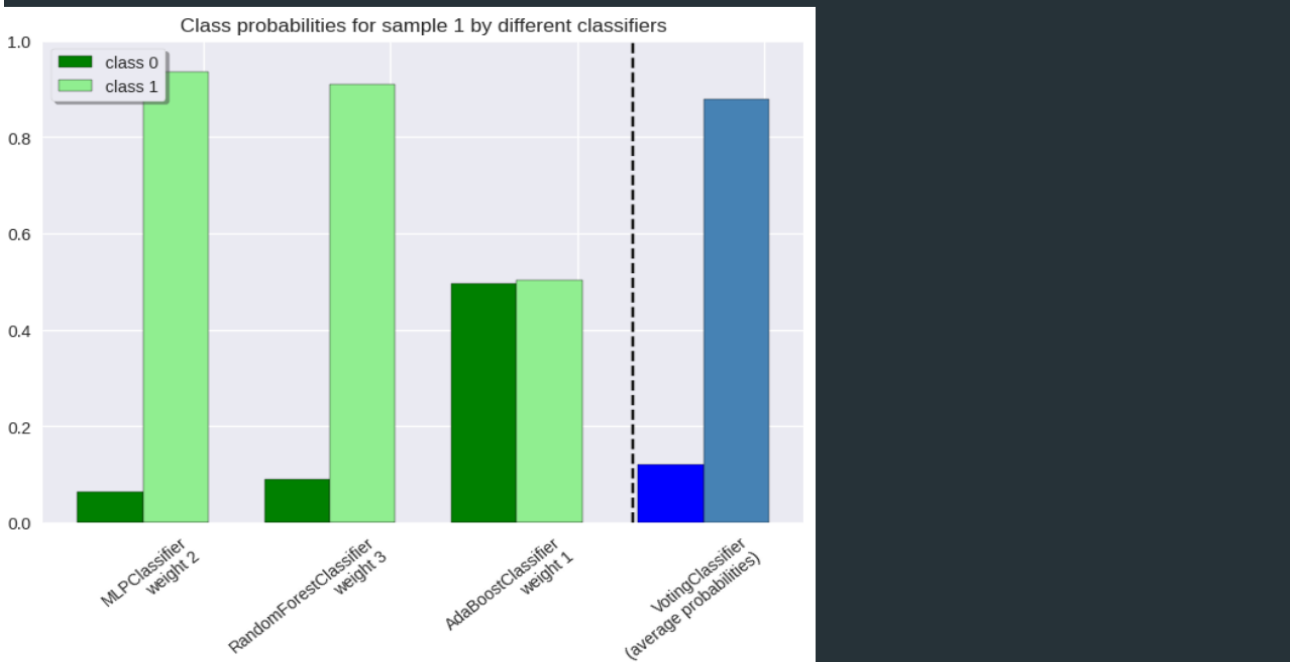
- Voting type was set to **'soft'**, meaning the class probabilities predicted by each best estimator are averaged, weighted by their importance to make final prediction.
- To prioritize best-performing models based on their performance, different weight configurations were explored to ensure that models like **RandomForestClassifier** have a more significant influence on the final prediction: **[1, 2, 1], [2, 2, 1], [1, 3, 1], [2, 3, 1], [3, 3, 1]**. Chosen weights assign the **highest importance** to the **RandomForestClassifier**, followed by **MLPClassifier** (can have the same level of importance) and **AdaBoostClassifier** (the lowest importance).
- **HalvingRandomSearchCV** was used to find the best combination of weights by iteratively narrowing down the best combinations.

```python
print('Best Soft Voting Weights:', voting_searcher.best_params_['weights'])
```

```
Best Soft Voting Weights: [2, 3, 1]
```

The weights **[2, 3, 1]** were optimized to leverage the strengths of **RandomForest** the most, followed by **MLPClassifier** and **AdaBoostClassifier**. This configuration reflects the models' performances and their complementary strengths.

```python
probas = [
    searcher.best_estimator_.predict_proba(X_test) # Predict class probabilities for all classifiers
    for name, searcher in {**searchers, 'VotingClassifier': voting_searcher}.items()
]
class0_0, class1_0 = [pr[0, 0] for pr in probas], [pr[0, 1] for pr in probas] # Get class probabilities for first sample
ind, width = np.arange(4), 0.35 # Group positions & Bar Width
p1 = plt.bar(ind, np.hstack(([class0_0[:-1], [0]])), width, color='green', edgecolor='k') # Bars for classifier 1-3
p2 = plt.bar(ind + width, np.hstack(([class1_0[:-1], [0]])), width, color='lightgreen', edgecolor='k')
p3 = plt.bar(ind, [0, 0, 0, class0_0[-1]], width, color='blue', edgecolor='k') # Bars for VotingClassifier
p4 = plt.bar(ind + width, [0, 0, 0, class1_0[-1]], width, color='steelblue', edgecolor='k')

plt.axvline(2.8, color='k', linestyle='dashed')
plt.xticks(ind + width)
plt.xticks(ind + 0.5, [*[
    f'{name}\nweight {weight}' for name, weight in zip(searchers, voting_searcher.best_params_['weights'])],
    'VotingClassifier\n(average probabilities)' # Plot annotations
], rotation=40, ha='right')
plt.ylim([0, 1])
plt.title('Class probabilities for sample 1 by different classifiers')
plt.legend([p1[0], p2[0]], ['class 0', 'class 1'], loc='upper left', frameon=True, shadow=True)
plt.show()
```



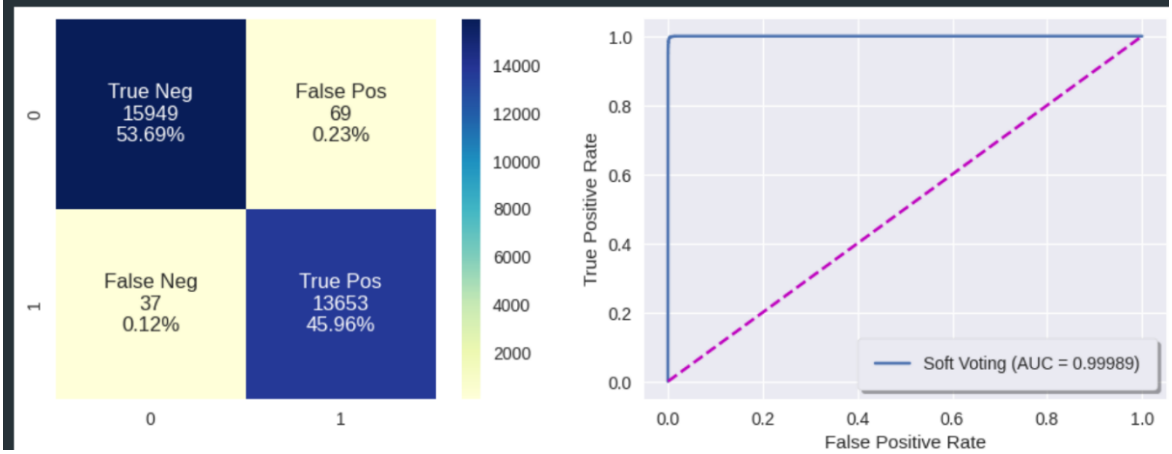
The plot illustrates how the **soft VotingClassifier** combines the probabilities predicted by each individual classifier with the assigned weights.

- **MLPClassifier**: Weight of 2 slightly increases its influence.
- **RandomForestClassifier**: Weight of 3 gives it the most significant influence, consistent with its performance.
- **AdaBoostClassifier**: Weight of 1 provides balanced but lesser influence.

Best Estimator	class 0	class 1
MLPClassifier	**2** * 0.07	**2** * 0.93
RandomForestClassifier	**3** * 0.10	**3** * 0.90
AdaBoostClassifier	**1** * 0.48	**1** * 0.52
weighted average	**0.15**	**0.85**

```python
y_pred = best_voting_clf.predict(X_test)
y_probs = best_voting_clf.predict_proba(X_test)
print(classification_report(y_test, y_pred, digits=5))

plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plot_cf_matrix(confusion_matrix(y_test, y_pred))

plt.subplot(1, 2, 2)
fpr, tpr, _ = roc_curve(y_test, y_probs[:, 1])
plt.plot(fpr, tpr, label=f'Soft Voting (AUC = {auc(fpr, tpr):.5f})')
plt.plot([0, 1], [0, 1], 'm--')

plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc='lower right', frameon=True, shadow=True, borderpad=1)
plt.show()
```

```
              precision    recall  f1-score   support

           0    0.99769   0.99569   0.99669     16018
           1    0.99497   0.99730   0.99613     13690

    accuracy                        0.99643     29708
   macro avg    0.99633   0.99649   0.99641     29708
weighted avg    0.99643   0.99643   0.99643     29708
```



- **Precision**: Achieved the **highest** precision scores for both classes with a slight improvement over individual classifier. **Class 1** (the minority class) shows high precision, indicating fewer **FP**.
- **Recall**: Also the **highest**, particularly for **Class 1**, reflecting fewer **FN**.
- **F1-Score**: Marginally higher than those of the individual classifiers, demonstrating a better balance between precision and recall.

There are **16,018** stars with **A** type and the model correctly recognizes **15,949** stars (**TN**). There are only **69** stars wrongly predicted to be **B** (**FP**). Similiarly, the test set has **13,690** stars with B type and only **37** of them are mistakenly identified as **A** (**FN**) and **13,653** stars are correctly identified as **B** (**TP**).

- **TP** and **TN**: Correctly identifies a higher number of instances for both classes, indicating stronger performance.

- **FP** and **FN**: Reduced values. The **FP** and **FN** rates are the **lowest** for **soft VotingClassifier,** showing its ability to minimize misclassifications compared to individual models. This reflects improved overall model performance and reliability.
- **ROC-AUC**: Achieved an **AUC** of **0.99989**, slightly higher than individual classifiers. This further confirmed improved **TP** rates and reduced **FP** rates, indicating its superior performance in distinguishing between classes.

5. Predictions on Unknown Dataset

The final step involved using the **Soft VotingClassifier**, which was identified as the best-performing model, to make predictions on an unknown dataset.

```
data_unknown = pd.read_csv('dataGaia_AB_unknown.csv', na_values=missing_values)
data_unknown.set_index('ID', inplace=True)
data_unknown = data_unknown.loc[:, ~data_unknown.columns.str.contains('^Unnamed|Source')] # Drop 3 unnecessary columns
data_unknown
```

ID	RA_ICRS	DE_ICRS	Plx	PM	pmRA	pmDE	Gmag	e_Gmag	BPmag	e_BPmag	...	BP-G	G-RP	pscol	Teff	Dist	
119089	64.085883	46.209497	0.3386	1.067	-0.841	-0.656	13.132349	0.002769	13.404948	0.002947	...	0.272599	0.444520	NaN	11663.4	2031.7985	2.8
24912	14.701590	67.754101	1.6877	3.527	-1.976	-2.921	12.781258	0.002763	13.285511	0.002884	...	0.504253	0.678857	NaN	7661.9	615.7902	1.7
6264	55.953330	38.656112	0.8378	8.216	6.574	-4.927	12.053966	0.002766	12.250828	0.002859	...	0.196862	0.331098	NaN	7820.4	1254.6984	2.9
81903	91.211868	-14.372262	1.1175	12.945	7.306	10.687	11.559693	0.003260	11.685955	0.005807	...	0.126262	0.251504	NaN	8984.8	890.9888	2.2
110917	75.949263	40.560254	0.9456	5.200	3.248	-4.061	15.013525	0.003206	15.266448	0.003034	...	0.252923	0.483553	NaN	10195.4	5160.0750	3.3
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
109624	72.354966	37.351363	0.1704	0.771	-0.517	-0.572	16.360239	0.002814	16.897928	0.006783	...	0.537689	0.723034	NaN	10047.2	3589.4248	1.9
19620	33.694149	58.156781	0.6546	3.614	2.664	-2.442	14.314589	0.002887	14.760225	0.003061	...	0.445637	0.658537	NaN	6897.3	2074.6511	2.9
145075	37.642425	56.450733	0.4109	1.440	-0.381	-1.388	13.153332	0.002774	13.330101	0.002879	...	0.176769	0.315199	NaN	12632.8	2188.9338	2.4
166955	21.766618	63.949002	0.2576	1.099	-1.093	-0.118	17.085732	0.002834	17.738290	0.007450	...	0.652559	0.823750	NaN	9675.2	3508.0160	1.6
157410	29.719213	58.405833	0.8439	1.167	-1.062	0.484	12.434747	0.002760	12.647573	0.002826	...	0.212827	0.357500	NaN	9902.5	1282.3630	2.3

37135 rows × 25 columns

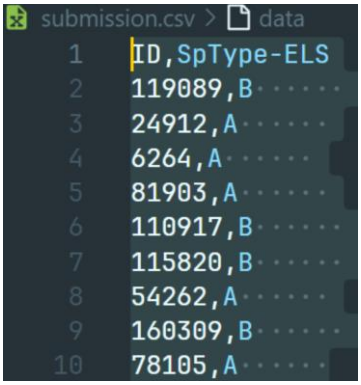
This dataset was preprocessed in the same manner as the training data to ensure consistency and reliability in predictions. This includes imputing missing values, winsorizing outliers, PCA, and feature scaling.

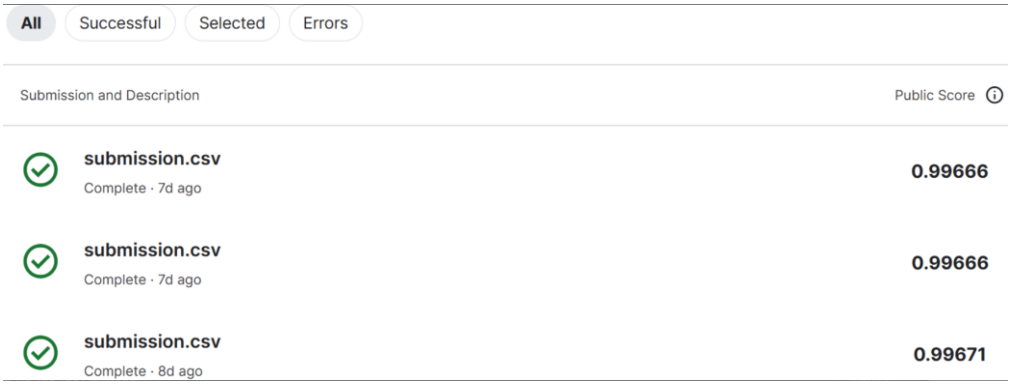
```
data_unknown.drop(columns=removed_columns, inplace=True, errors='ignore')
data_unknown = imputer.transform(data_unknown)
data_unknown, outliers_summary = winsorize_outliers(data_unknown, limits=[0.05, 0.05])
data_unknown = pca_grouping_test(data_unknown, pca_dict, pca_features)
data_unknown[robust_cols] = robust_scaler.transform(data_unknown[robust_cols])
data_unknown[standard_cols] = standard_scaler.transform(data_unknown[standard_cols])
```

The preprocessed unknown dataset was passed to the optimized **Soft VotingClassifier** to make predictions. The predicted labels were formatted according to the required submission format for Kaggle, including inversing transformed to their original categorical format (A or B) with the addition of spaces to match the submission specification. The predictions were saved to a CSV file & submitted to Kaggle.

```python
y_pred_unknown = best_voting_clf.predict(data_unknown)
y_pred_unknown = label_encoder.inverse_transform(y_pred_unknown)
y_pred_unknown = np.char.add(y_pred_unknown.astype('<U6'), '      ') # Add spaces to match the submission format

pd.DataFrame(
    y_pred_unknown,
    index=data_unknown.index, columns=['SpType-ELS']
).to_csv('submission.csv')
```

The submitted predictions achieved a public score of **0.99671** on Kaggle, indicating the model's excellent performance and generalization capability on unseen data. Additionally, what might make my submission different from others is that I only need **16 features** to acquire a near-perfect score.





IV. Justify the Classifier Selected

After an extensive evaluation of various classifiers, the best model selected is the **Soft VotingClassifier**. It demonstrates the power of ensemble methods in achieving superior performance (particularly in complex decision boundaries) by averaging the predicted probabilities from individual well-tuned classifiers using a **Soft Voting** mechanism with optimized weights to provide a more robust and reliable classifier with most accurate predictions: **RandomForestClassifier**, with the highest weight, contributes most significantly, while **MLPClassifier** and **AdaBoostClassifier** provide complementary strengths.

	Pros	Cons	Why VotingClassifier?
MLP Classifier	- Excellent at capturing complex, non-linear patterns in data due to its neural network architecture. - Can adapt to various problems with different	- Prone to *overfitting* without careful tuning and regularization, especially with high-dimensional data, as seen in the high training	- **MLP** showed excellent performance but was prone to slight *overfitting* and required extensive computational resources.

	configurations of hidden layers and neurons. - Performed well in terms of Precision and Recall.	scores but slightly lower validation scores from **iniital testing** phase. - Computationally intensive and requires significant tuning.	- **VotingClassifier** mitigated these issues by combining with more stable models like **RandomForest**.
Random Forest Classifier	- Handle many features and data points effectively with high resilience to overfitting due to averaging multiple Decision Trees. - Provide insights into the importance of features, aiding in understanding the model's decisions. - **class_weight** adjustments help manage class imbalance effectively.	- Computationally expensive with a large number of trees and features. - May not capture intricate patterns as effectively as Neural Networks or Boosting methods.	- Although **Random Forest** also showed excellent performance & was a top contender, **VotingClassifier** outperformed by combining its robustness with the strengths of **MLP** and **AdaBoost**, leading to slightly better overall metrics.
AdaBoost Classifier	- Improve and combines weak learners to form a strong learner, improving performance iteratively. - Adjusts to the difficulty of the task by focusing on misclassified instances in each iteration. - Tends to generalize well to unseen data. - Boosting approach improves performance by focusing on hard-to-classify instances.	- Require careful tuning to avoid *overfitting* & can overfit noisy dataset, especially with too many iterations or high learning rates. - Sensitive to outliers and noisy data, which can degrade performance. - Boosting can be computationally intensive with many iterations.	- **AdaBoost** performed well individually. However, its potential sensitivity to noise and outliers was a concern. - **VotingClassifier** balanced **AdaBoost**'s strengths with the robustness of **Random Forest** and **MLP**, leading to more stable performance

The **Soft VotingClassifier** outperformed individual best estimators from the hyperparameter tuning step, with consistently the highest metrics across all evaluation criteria. This indicates its superior ability to generalize well on unseen data and handle class imbalance, addressing potential overfitting and underfitting issues.

- **F1-Score**: Achieved the highest F1-score among all tested models, demonstrating a balanced performance between Precision and Recall.
- **Precision and Recall**: Show superior Precision and Recall, particularly in handling the minority class, which is crucial for imbalanced datasets.
- **ROC-AUC**: Near-perfect AUC of **0.99989**, indicating excellent discriminatory power between classes.
- **Mitigation of Overfitting**: Ensemble method helps reduce overfitting by averaging predictions, which smooths out the noise & biases present in individual classifiers.
- **Error Reduction**: **VotingClassifier** demonstrated superior handling of classification errors compared to individual classifiers, particularly **FP** and **FN**, which are critical in high-stakes decision-making scenarios, leading to more reliable and stable predictions.

By combining the predictive power of these models, this leads to improved overall performance while smoothing out their individual biases and weaknesses. With the complex and high-dimensional Gaia's dataset, the **Soft VotingClassifier** effectively combined the deep learning capabilities of **MLPClassifier** to capture complex patterns, **RandomForestClassifier**'s robustness, and boosting strength of **AdaBoostClassifier** focusing on hard-to-classify instances to work together & enhance prediction reliability. This approach exemplifies the principle of "wisdom of the crowd", where the collective decision-making of several models leads to better outcomes than any single model.