Feature Selection and Modelling

Table of Contents

Feature Selection

Feature selection is the process of reducing the number of input variables when developing a predictive model.

It is desirable to reduce the number of input variables to both reduce the computational cost of modelling and, in some cases, to improve the performance of the model.

Statistical-based feature selection methods involve evaluating the relationship between each input variable and the target variable using statistics and selecting those input variables that have the strongest relationship with the target variable. These methods can be fast and effective, although the choice of statistical measures depends on the data type of both the input and output variables.

Variance Thresholding

If the variance is low or close to zero, then a feature is approximately constant and will not improve the performance of the model. In that case, it should be removed.

Variance will also be very low for a feature if only a handful of observations of that feature differ from a constant value.

What we can do is set a threshold and drop features with low variance

Anova Test

Analysis of variance (ANOVA) is a statistical technique that is used to check if the means of two or more groups are significantly different from each other. ANOVA checks the impact of one or more factors by comparing the means of different samples.

If we had categorical variables, we would do another test called the $\chi 2$ test. Since we have all numeric features, we do the ANOVA test.

Recursive Feature Elimination

Recursive Feature Elimination selects features by recursively considering smaller subsets of features by pruning the least important feature at each step. Here models are created iteratively and, in each iteration, it determines the best and worst performing features and this process continues until all the features are explored. Next ranking is given on each feature based on their elimination order. In the worst case, if a dataset contains N number of features RFE will do a greedy search for $N2$N2 combinations of features.

Feature selection using Random Forest

Feature selection using Random Forest comes under the category of Embedded methods. Embedded methods combine the qualities of filter and wrapper methods. They are implemented by algorithms that have their own built-in feature selection methods. Some of the benefits of embedded methods are:

1. They are highly accurate.
2. They generalize better.
3. They are interpretable

Here is a summary of all the feature selection methods and the features selected

Methods	Features Selected
Variance Thresholding (threshold = 1)	['Area', 'Perimeter', 'MajorAxisLength', 'MinorAxisLength', 'AspectRation', 'Eccentricity', 'ConvexArea', 'EquivDiameter', 'Extent', 'roundness', 'Compactness', 'ShapeFactor1', 'ShapeFactor2', 'ShapeFactor3']
ANOVA F-test	['Area', 'Perimeter', 'MajorAxisLength', 'MinorAxisLength', 'ConvexArea', 'EquivDiameter', 'ShapeFactor1', 'ShapeFactor2']
Recursive Feature Elimination (estimator = DecisionTreeClassifier)	['Perimeter', 'MajorAxisLength', 'MinorAxisLength', 'roundness', 'Compactness', 'ShapeFactor1', 'ShapeFactor3', 'ShapeFactor4']
Using RandomForest feature importance	['Perimeter', 'MajorAxisLength', 'MinorAxisLength', 'AspectRation', 'EquivDiameter', 'Compactness', 'ShapeFactor1', 'ShapeFactor3']

Modelling

```
from pycaret.classification import *
from sklearn import metrics
from sklearn.model_selection import train_test_split

import seaborn as sns
from joblib import dump, load
import json
import os

sns.set(rc={"figure.figsize": (10, 8)}, font_scale=1.25)

df = pd.read_csv("./DryBeanDataset/Dry_Bean_Dataset.csv").sample(frac=1).reset_index(drop=True)

df.head()
```

Experiment without transformed data

Docs: PyCaret

We setup a pycaret experiment. The parameters are:

- data: df

- `target`: `Class`
- `normalize`: Normalizes all the numeric features using method mentioned using `normalize_method` if set to `True`
- `transformation`: Applies `yeo-johnson` transformation or method mentioned using `transform_method` if set to `True`
- `fix_imbalance`: Fixes imbalance using SMOTE or method mentioned using `imbalance_method` if set to `True`

```
exp = setup(
    data=df,
    target='Class',
    train_size=0.7,
    experiment_name='baseline_without_transforms',
    remove_perfect_collinearity=False
)
```

```
<pandas.io.formats.style.Styler at 0x1ebd4f4ff10>
```

Comparing base-line models

Calling the `compare_models()` is going to fit all classification models for our data

```
%%time
best_model = compare_models()
best_model
```

```
<pandas.io.formats.style.Styler at 0x1ebd4a1bd00>
```

```
Wall time: 2min 5s
```

```
LGBMClassifier(boosting_type='gbdt', class_weight=None, colsample_bytree=1
.0,
               importance_type='split', learning_rate=0.1, max_depth=-1,
               min_child_samples=20, min_child_weight=0.001, min_split_gai
n=0.0,
               n_estimators=100, n_jobs=-1, num_leaves=31, objective=None,
               random_state=8669, reg_alpha=0.0, reg_lambda=0.0, silent='w
arn',
               subsample=1.0, subsample_for_bin=200000, subsample_freq=0)
```

The F1 here is the weighted f1 we are using as a metric. So, that's good, we can also pass a custom metric

Obvservation

- Light Gradient Boosting Classifier performs the best among all the baselines, without us doig any transforming or feature selection at a F1 of approx ~ 93

This is pretty good, let's see if we can stretch it further using tuning the model.

Individual Estimators and tuning them

We can see there's not much difference between gradient boosting and the LGBM Classifier. WE will start with LGBM as it's faster to train

```
def plot(estimator, plot_type, dst):
    res = plot_model(
        estimator=estimator,
        plot=plot_type,
        save=True
    )

    os.rename(res, dst)
    for file in os.listdir("."):
        if file.endswith('.png'):
            os.remove(file)

def clean_params(params):
    d = {}
    for key, value in params.items():
        d[key.replace('actual_estimator__', "")] = value

    return d

def save(model, tuner=None):
    if tuner is not None:
        dump(
            model,
            filename=f"./ML_models/PC_{model.__class__.__name__}_{tuner.__
class__.__name__}.model"
        )
        with open(
            f'./ML_results/PC_{model.__class__.__name__}_{tuner.__class__.
__name__}_params.json',
            mode='w'
        ) as f:
            json.dump(clean_params(tuner.best_params_), fp=f)

        print(f"Model saved at: ./ML_models/PC_{model.__class__.__name__}_
{tuner.__class__.__name__}.model")
        print(f"Tuner saved at: ./ML_results/PC_{model.__class__.__name__}
_{tuner.__class__.__name__}_params.json")

    else:
        dump(
            model,
            filename=f"./ML_models/PC_{model.__class__.__name__}_baseline.
model"
        )
        print(f"Model saved at: ./ML_models/PC_{model.__class__.__name__}_
baseline.model")
```

Light Gradient Boosting

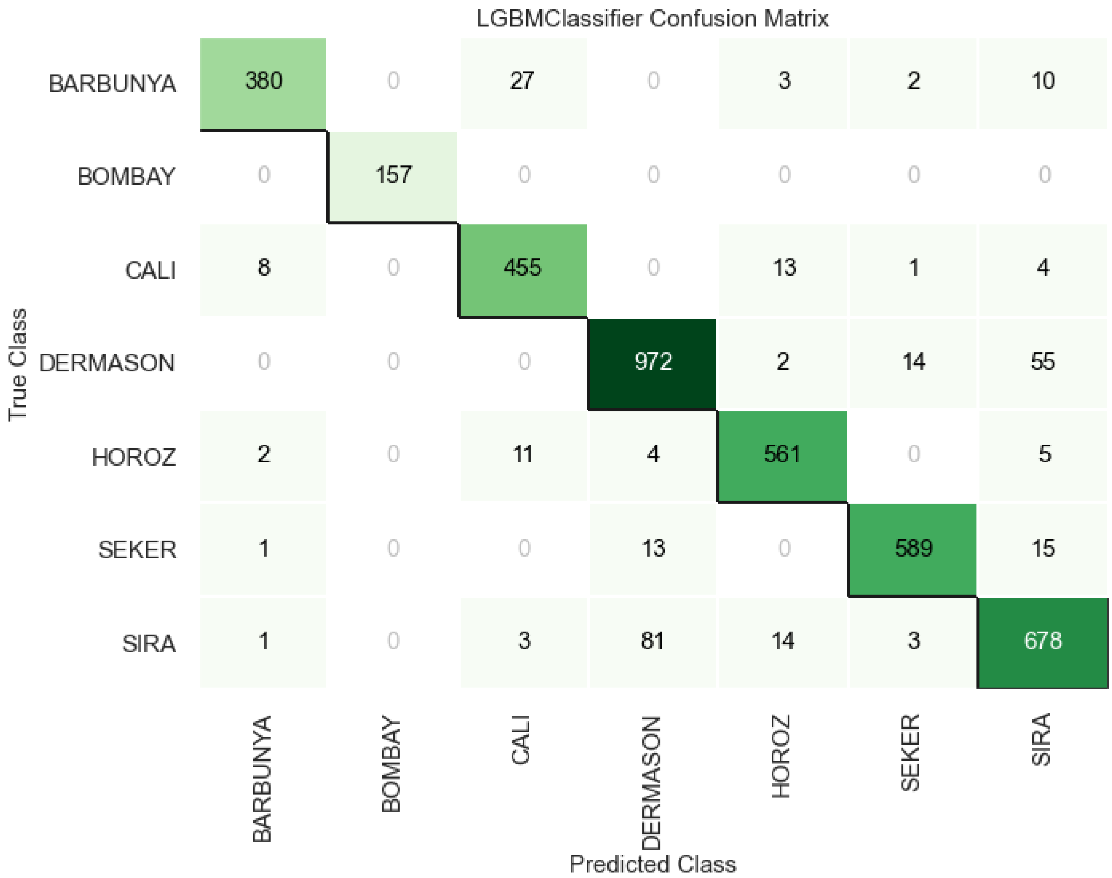
```
%%time
lgbm = create_model('lightgbm')
```

```
<pandas.io.formats.style.Styler at 0x1ebd4a3a5b0>
```

```
Wall time: 6.24 s
```

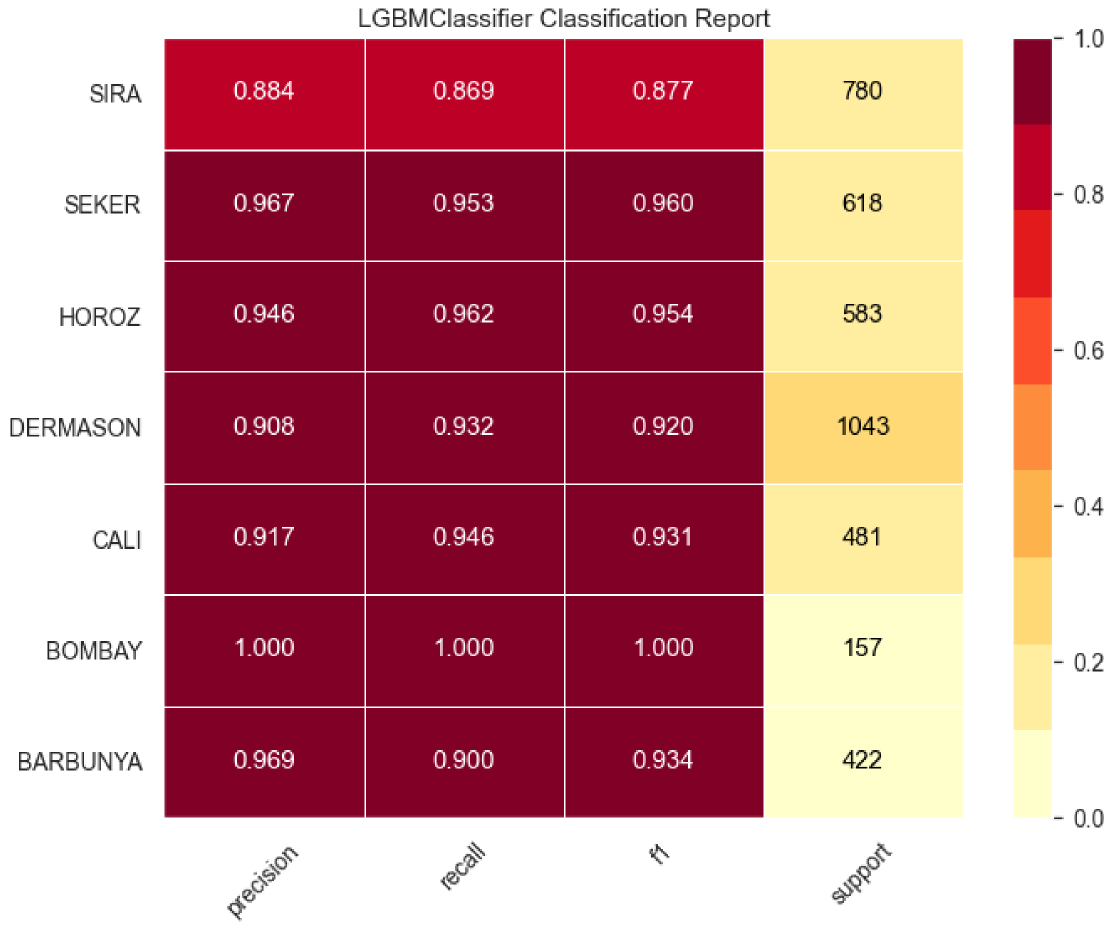
Plotting different plots like confusion matrix and auc is also very easy as simple as 1 line of code. We look at few plots, to asses performance

```
plot_model(lgbm, plot='confusion_matrix')
```



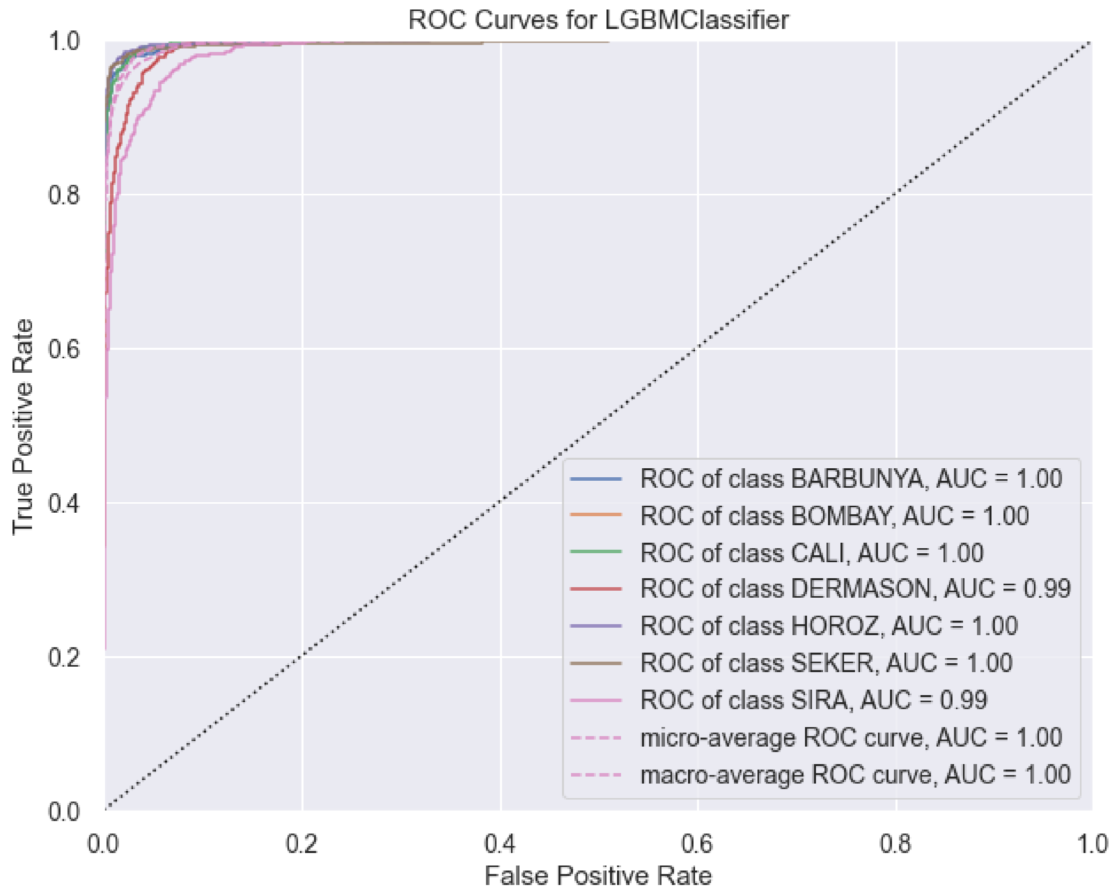

```
# plot(estimator=lgbm, plot_type='confusion_matrix', dst=f"./ML_results/CF_{lgbm.__class__.__name__}.png")
```

```
plot_model(lgbm, plot='class_report')
```




```
plot(estimator=lgbm, plot_type='class_report', dst=f'./ML_results/ClassRep
ort_{lgbm.__class__.__name__}.png')

plot_model(estimator=lgbm, plot='auc')
```




```
plot(lgbm, plot_type='auc', dst=f"./ML_results/AUC_{lgbm.__class__.__name__}.png")
```

```
save(model=lgbm)
```

```
Model saved at: ./ML_models/PC_LGBMClassifier_baseline.model
```

Obvservation

- The baseline lightgbm performs well with an f1 of approx ~ 93
- Our model seems to be confused between DERMASON and SIRA varieties
- Our precision, recall and f1 for each class is more than 86, which is also a good indication

Tuning The LightGBM

Tuning the `LightGBMClassifier`. We can do Grid-search, Random-search as these are the good old hyper paramter tuning methods. But there's a more efficient tuning method using Bayesian Hyperparamter tuning. Here's a one line summary of what bayesian search is:

> Build a probability model of the objective function and use it to select the most promising hyperparameters to evaluate in the true objective function.

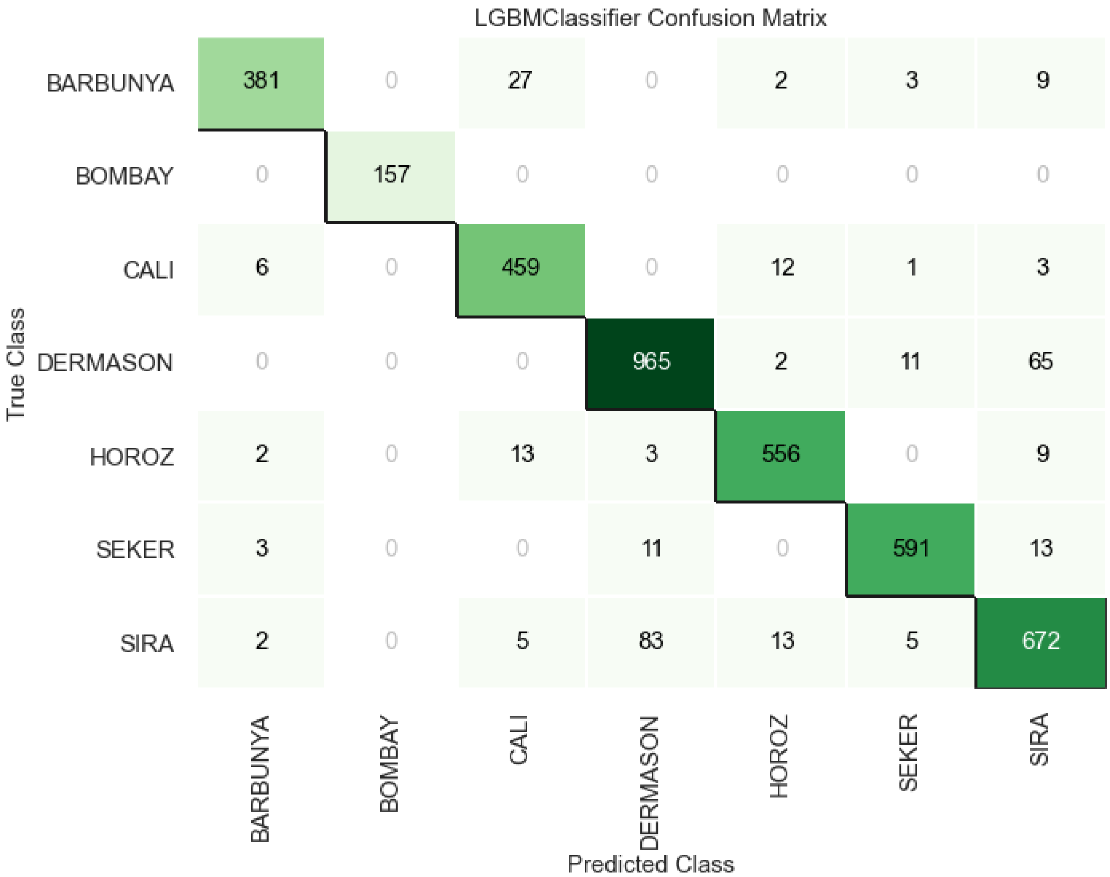
```
%%time
tuned_lgbm, tuner = tune_model(
    estimator=lgbm,
    search_library="scikit-optimize",
```

```
    n_iter=25,
    optimize='f1', return_tuner=True
)
```

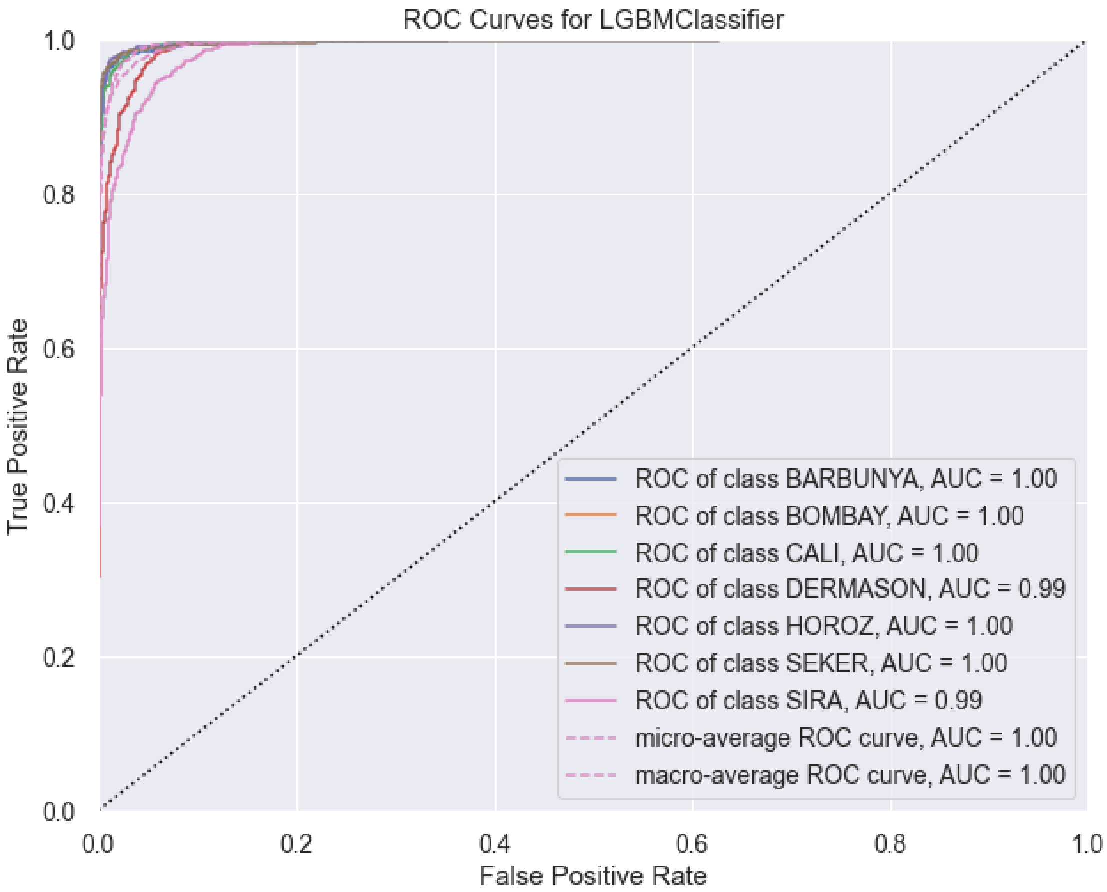
<pandas.io.formats.style.Styler at 0x1ebd6374c70>

Wall time: 2min 17s

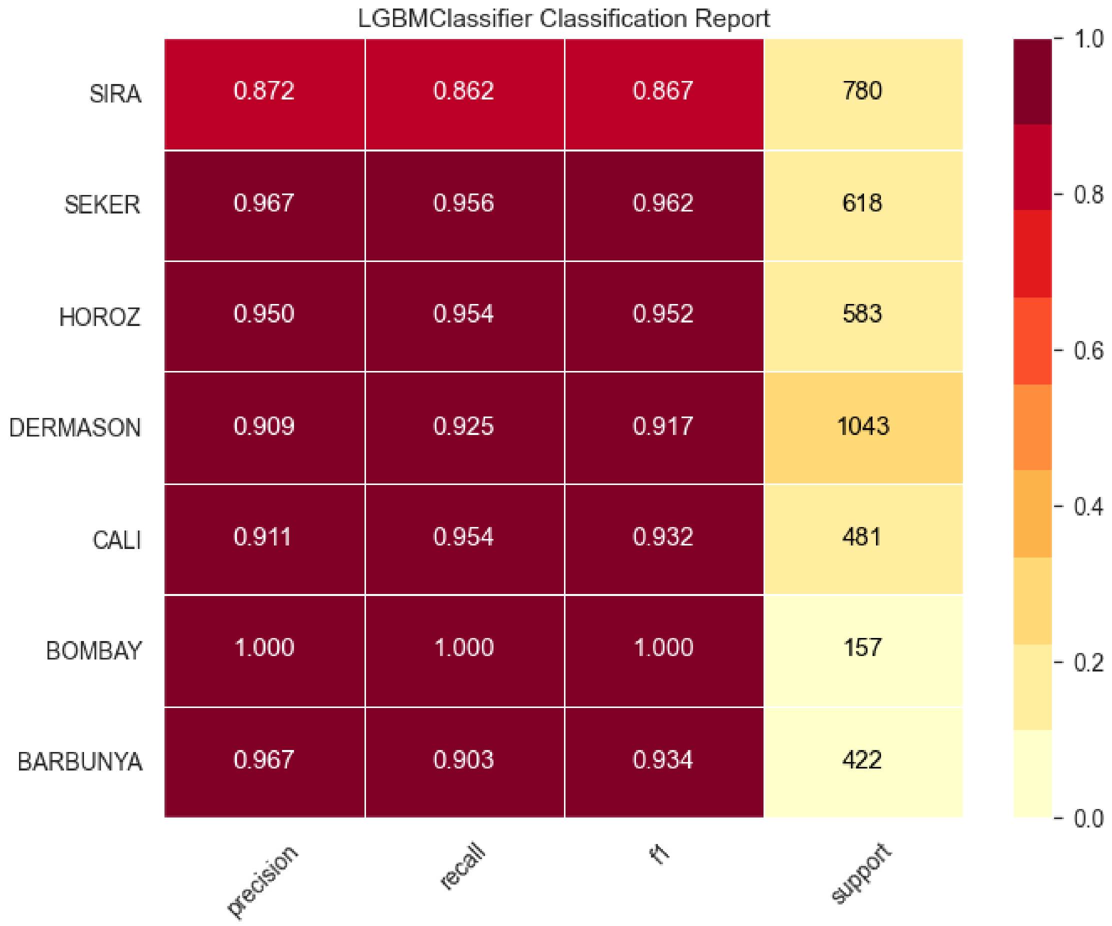
```
plot_model(tuned_lgbm, plot='confusion_matrix')
```




```
plot_model(tuned_lgbm, plot='auc')
```




```
plot_model(tuned_lgbm, plot='class_report')
```




```
plot_model(tuned_lgbm, plot='parameter')
```

	Parameters
boosting_type	gbdt
class_weight	None
colsample_bytree	1.0
importance_type	split
learning_rate	0.07351087677623395
max_depth	-1
min_child_samples	8
min_child_weight	0.001
min_split_gain	0.308079603643812
n_estimators	132
n_jobs	-1
num_leaves	109
objective	None
random_state	8669
reg_alpha	2.878849039397276
reg_lambda	2.7700803937357488e-08
silent	warn
subsample	1.0
subsample_for_bin	200000
subsample_freq	0
bagging_fraction	0.740705467313804
bagging_freq	2
feature_fraction	0.768581357957563

We can access the search space easily too:

```
params = tuner.get_params()
params['search_spaces']
```

```
{'actual_estimator__num_leaves': Integer(low=2, high=256, prior='uniform', transform='normalize'),
 'actual_estimator__learning_rate': Real(low=1e-06, high=0.5, prior='log-uniform', transform='normalize'),
 'actual_estimator__n_estimators': Integer(low=10, high=300, prior='uniform', transform='normalize'),
 'actual_estimator__min_split_gain': Real(low=0, high=1, prior='uniform', transform='normalize'),
 'actual_estimator__reg_alpha': Real(low=1e-10, high=10, prior='log-uniform', transform='normalize'),
 'actual_estimator__reg_lambda': Real(low=1e-10, high=10, prior='log-uniform', transform='normalize'),
 'actual_estimator__feature_fraction': Real(low=0.4, high=1, prior='uniform', transform='normalize'),
 'actual_estimator__bagging_fraction': Real(low=0.4, high=1, prior='uniform', transform='normalize'),
 'actual_estimator__bagging_freq': Integer(low=0, high=7, prior='uniform', transform='normalize'),
 'actual_estimator__min_child_samples': Integer(low=1, high=100, prior='uniform', transform='normalize')}
```

```
tuner.best_params_
```

```
OrderedDict([('actual_estimator__bagging_fraction', 0.740705467313804),
             ('actual_estimator__bagging_freq', 2),
             ('actual_estimator__feature_fraction', 0.768581357957563),
             ('actual_estimator__learning_rate', 0.07351087677623395),
             ('actual_estimator__min_child_samples', 8),
             ('actual_estimator__min_split_gain', 0.308079603643812),
             ('actual_estimator__n_estimators', 132),
             ('actual_estimator__num_leaves', 109),
             ('actual_estimator__reg_alpha', 2.878849039397276),
             ('actual_estimator__reg_lambda', 2.7700803937357488e-08)])
```

All of the class stuff we wrote in the previous notebook where we were doing custom tuning is now reduced to just a single line of code and we have full control over it. The default search-space provided in pycaret is good enough for tuning, but we also pass a custom grid like we did in our previous notebook

```
Model saved at: ./ML_models/PC_LGBMClassifier_BayesSearchCV.model
Tuner saved at: ./ML_results/PC_LGBMClassifier_BayesSearchCV_params.json
```

We can also let pycaret choose for us if we don't want to use bayesian search

```
tuned_lgbm_auto, tuner_auto = tune_model(
    estimator=lgbm,
    choose_better=True,
    optimize='f1', return_tuner=True
)
```

```
<pandas.io.formats.style.Styler at 0x1ebd4d9bc40>
```

```
tuner_auto.best_params_
```

```
{'actual_estimator__reg_lambda': 2,
 'actual_estimator__reg_alpha': 1e-07,
 'actual_estimator__num_leaves': 256,
 'actual_estimator__n_estimators': 70,
 'actual_estimator__min_split_gain': 0.9,
 'actual_estimator__min_child_samples': 91,
 'actual_estimator__learning_rate': 0.15,
 'actual_estimator__feature_fraction': 0.6,
 'actual_estimator__bagging_freq': 4,
 'actual_estimator__bagging_fraction': 1.0}
```

```
tuner_auto.__class__
```

```
sklearn.model_selection._search.RandomizedSearchCV
```

Obvservation

- Setting `choose_better=True`, it uses RandomSearchCV instead of BayesSearchCV
- There's no drastic difference between the two. Infact, random search is just randomly searching for the parameters.
- So, BayesSearch is better than random search in the sense that it uses a probability distribution rather than it just doing random search, which is more efficient as the probability guides the search

Results - 1

- The highest accuracy score mentioned in the paper which is 93.13 %. In the paper, they get to it through SVM with a polynomial kernel.
- We have reached an accuracy of 92.72 % with a LightGBM, which you can say is faster to train than the SVM whose time complexity would be Quadratic
- I have used no preprocessing or transformation or fixed the target imbalance, similar to the paper
- I have used all the 16 features

Other models mentioned in the paper

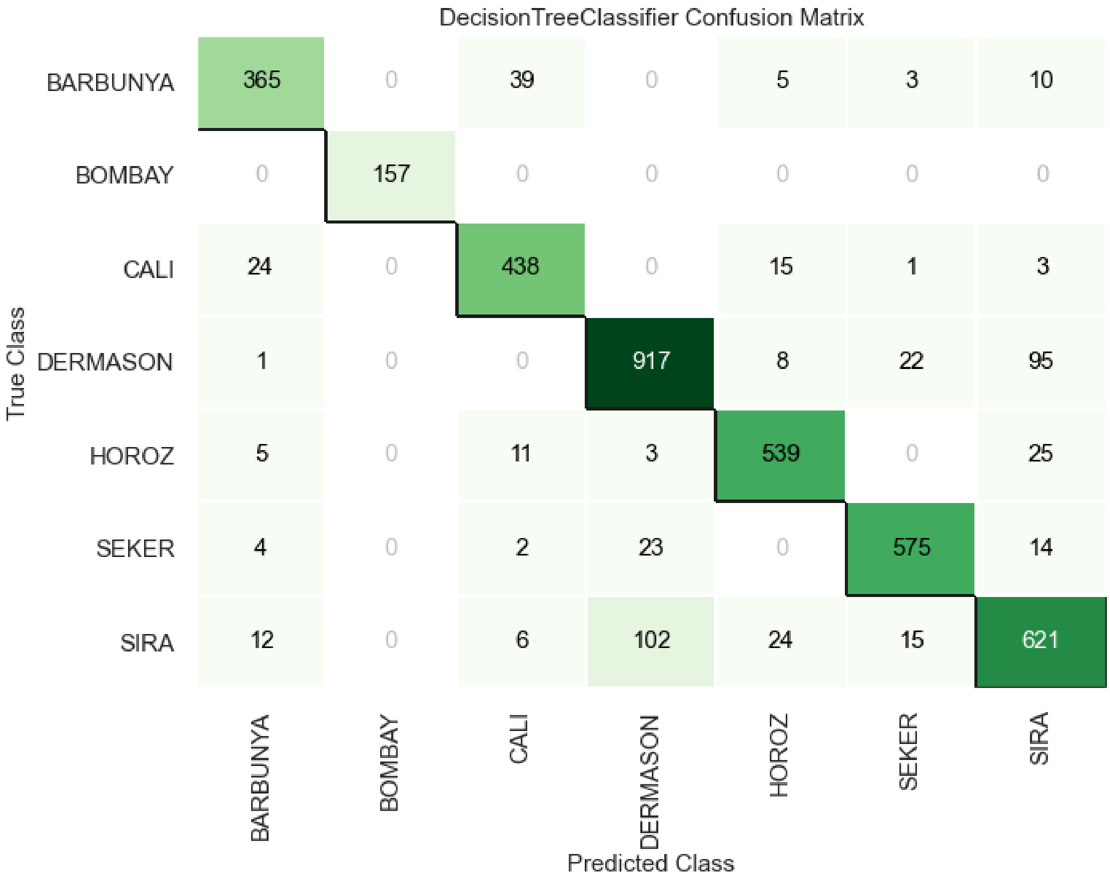
<u>Decision Tree</u>

```
%%time
dt = create_model('dt')
```

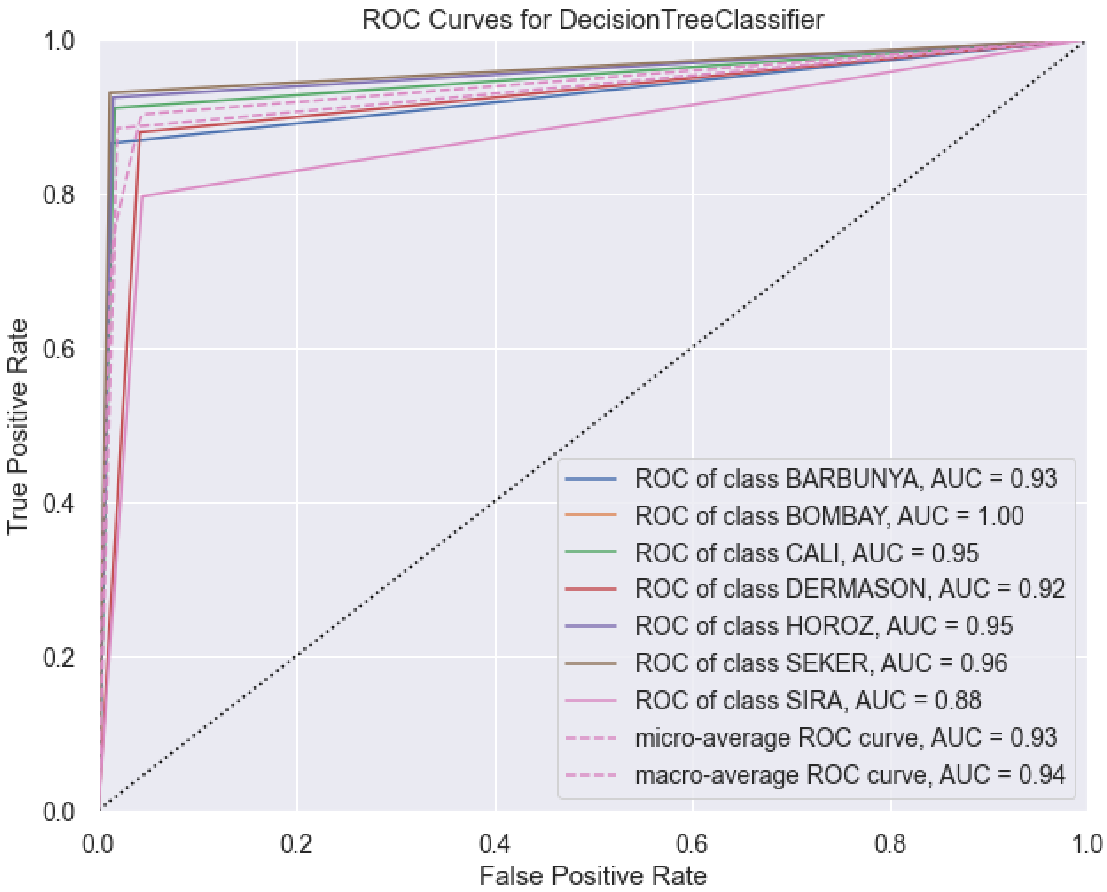
```
<pandas.io.formats.style.Styler at 0x1ebd4d9b640>
```

```
Wall time: 879 ms
```

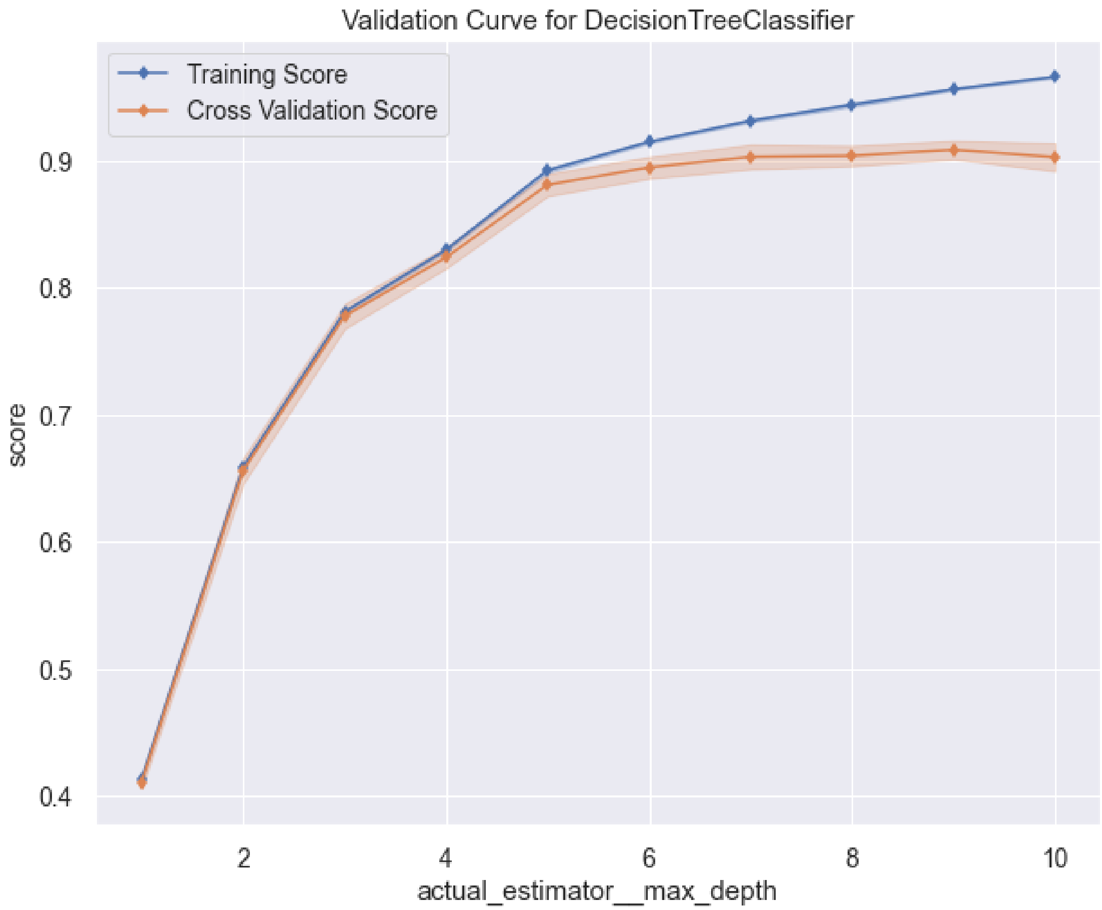
```
plot_model(dt, plot='confusion_matrix')
```




```
plot_model(dt, plot='auc')
```




```
plot_model(dt, plot='vc')
```

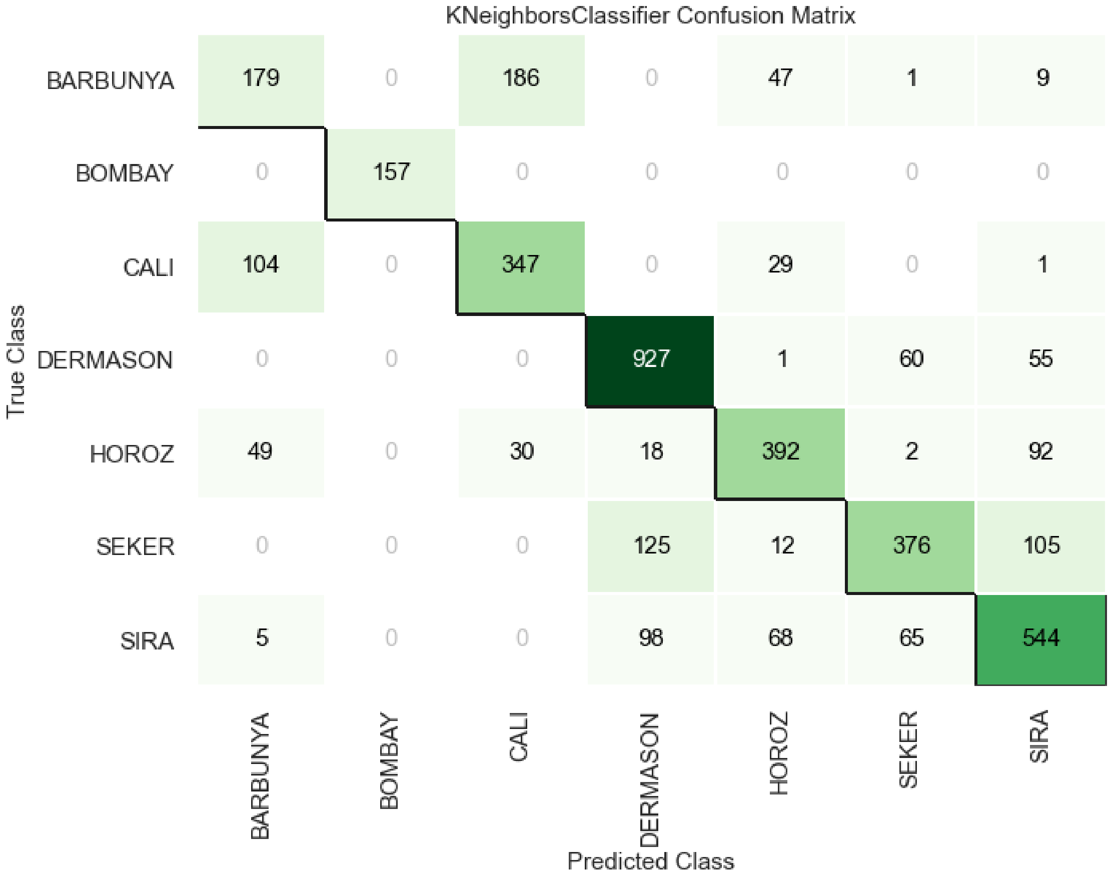


<u>KNN</u>

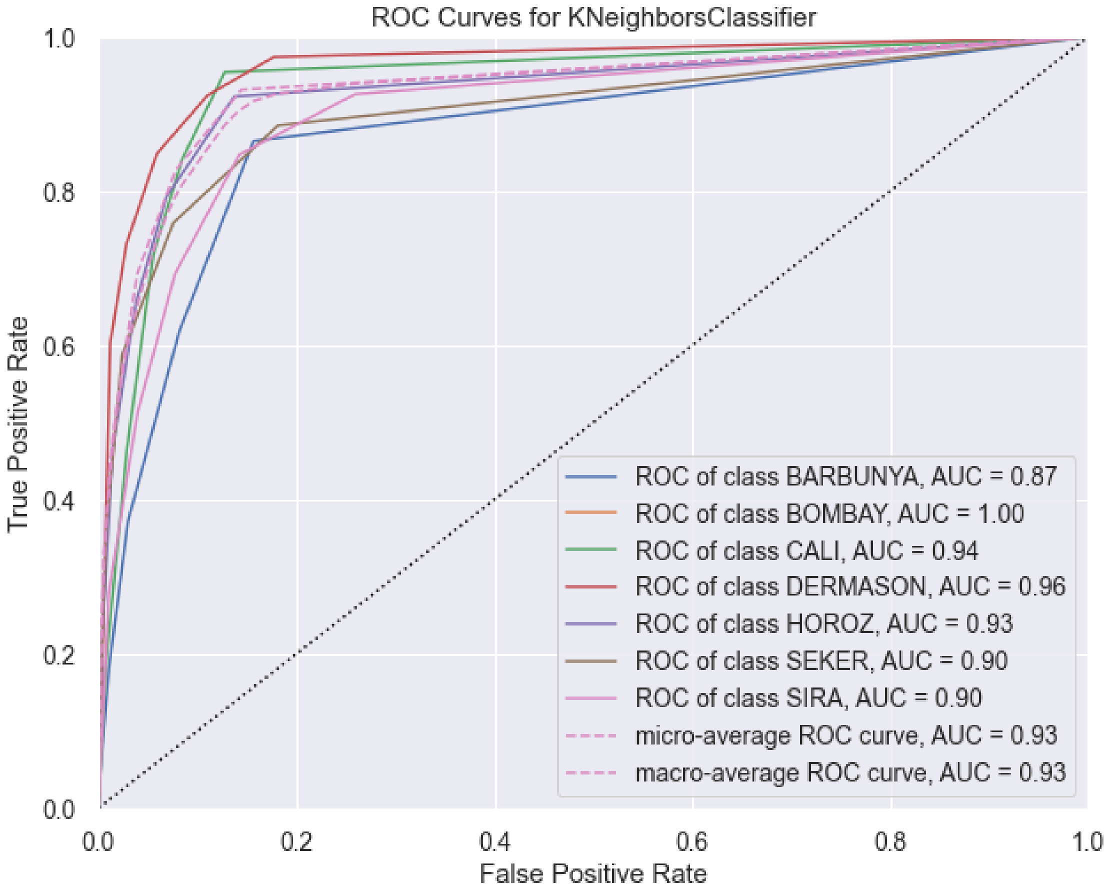
```
knn = create_model('knn')
```

```
<pandas.io.formats.style.Styler at 0x1ebd6147340>
```

```
plot_model(knn, plot='confusion_matrix')
```



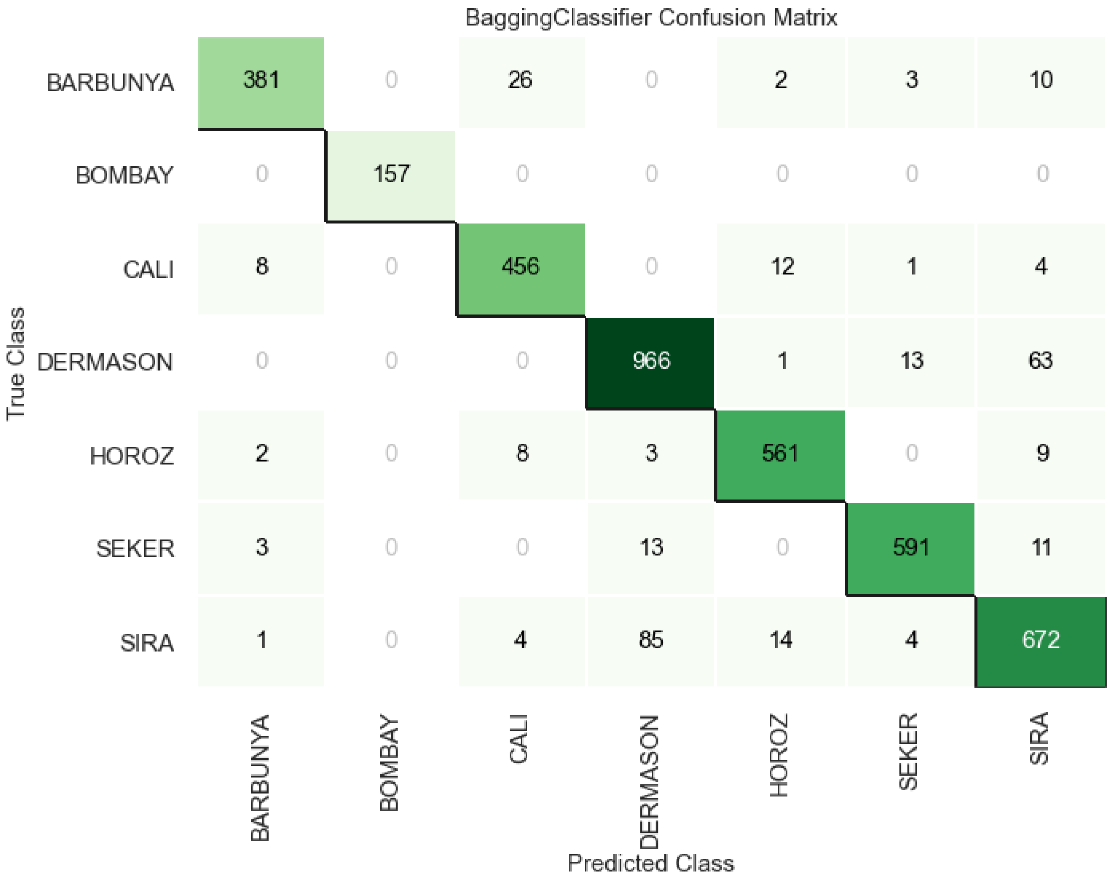
```
plot_model(knn, plot='auc')
```



Ensembling

Ensembled Light Gradient Boosting

```
ensembled_lgbm = ensemble_model(tuned_lgbm, optimize='f1')
<pandas.io.formats.style.Styler at 0x1ebd4d9bac0>
plot_model(ensembled_lgbm, plot='confusion_matrix')
```

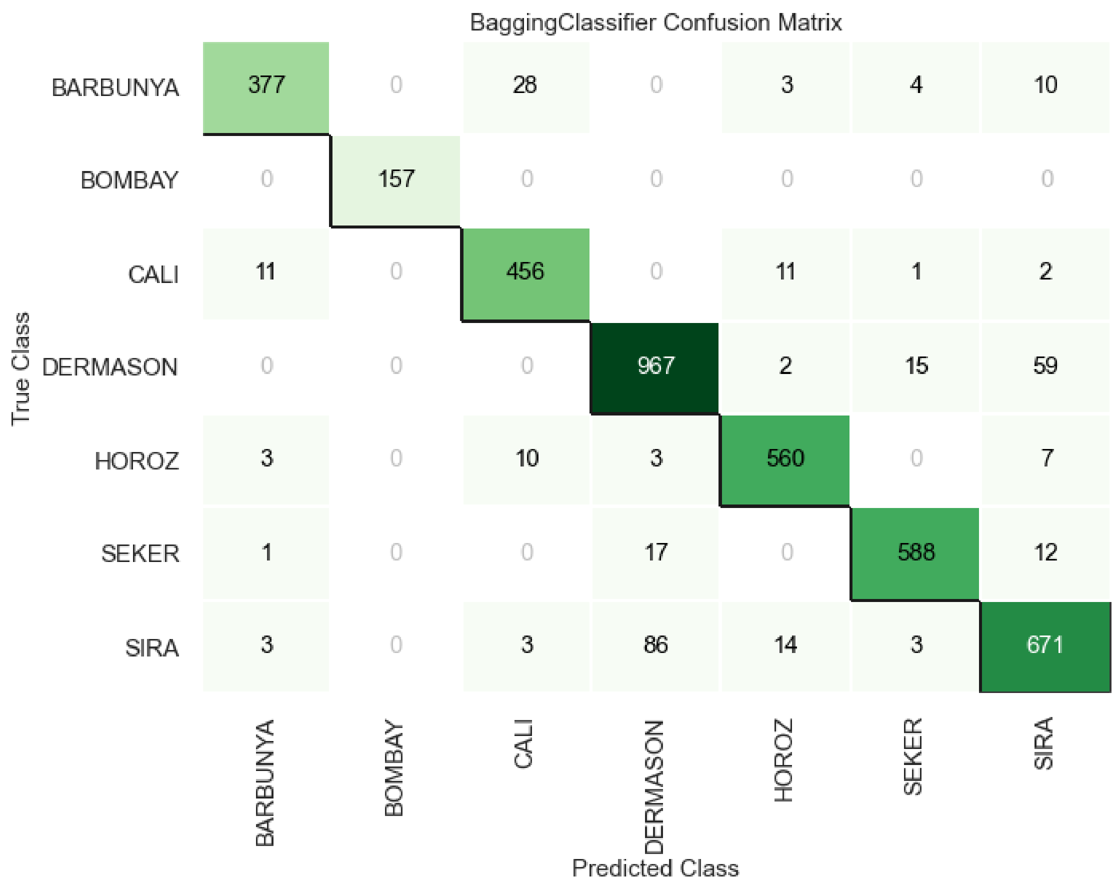


<u>Ensembled Decision Tree</u>

```
ensembled_dt = ensemble_model(dt, n_estimators=100, optimize='f1')
```

```
<pandas.io.formats.style.Styler at 0x1ebd6392910>
```

```
plot_model(ensembled_dt, plot='confusion_matrix')
```




```
save(model=ensembled_dt)
```

```
Model saved at: ./ML_models/PC_BaggingClassifier_baseline.model
```

Obvsesrvation

- Both the ensembled LightGBM and Decision tree, do not do well than out tuned LightGBm model.

Blending

We will try blending the tuned light gbm and ensembled decision tree model

```
blended_lgbm_dt = blend_models(estimator_list=[tuned_lgbm, ensembled_dt],
optimize='f1')
```

```
tuned_blended_lgbm_dt = tune_model(estimator=blended_lgbm_dt, search_libra
ry='scikit-optimize', optimize='f1')
```

Results - 2

- The highest accuracy score mentioned in the paper which is 93.13 %. In the paper, they get to it through SVM with a polynomial kernel.
- We have reached an accuracy of 92.78 % with a blended model of tuned lightgbm + ensembled decision tree,
- I have used no preprocessing or transformation or fixed the target imbalance, similar to the paper
- I have used all the 16 features

Experiment with transformed data

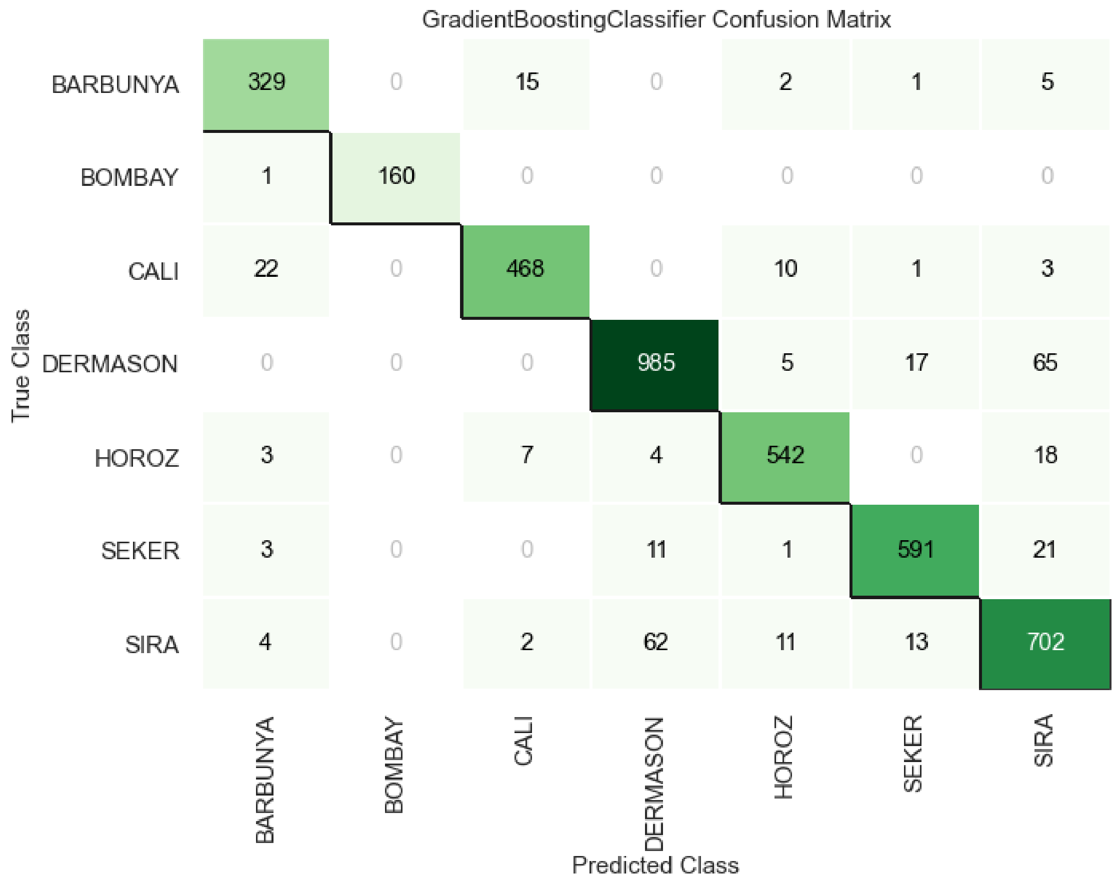
```
exp = setup(
    data=df,
    target='Class',
    train_size=0.7,
    experiment_name='baseline_with_transforms',
    remove_perfect_collinearity=False, fix_imbalance=True, normalize=True,
transformation=True
)
```

```
<pandas.io.formats.style.Styler at 0x1ebd4e6a0d0>
```

```
best_model = compare_models()
```

```
<pandas.io.formats.style.Styler at 0x1ebd639c9a0>
```

```
plot_model(best_model, plot='confusion_matrix')
```




```
%%time
tuned_lgbm_transformed, tuner_lgbm_transformed = tune_model(
    best_model,
    optimize='f1',
    search_library='scikit-optimize',
    return_tuner=True
)
```

Results - 3

- The highest accuracy score mentioned in the paper which is 93.13 %. In the paper, they get to it through SVM with a polynomial kernel.
- We have reached an accuracy of 92.9 % ~ 93 % with lightgbm after preprocessing the data by normalizing it and fixing the imbalance
- I have used all the 16 features

Conclusion

- The best model found was transformed data + Light Gradient Boosting
- We can also go with simple Light Gradient Boosting as the difference between them is not that significant

Git Repo with all the results and models: Diploma_thesis