Ensemble Learning: Random Forests

AAA-Python Edition

Plan

- 1- Random Forest
- 2- Extreme Random Forest
- 3- Balanced Random Forest
- 4- Feature Importance
- 5- Grid Search
- 6- Balancing by resampling

Concept

- A random forest is :

Bagging or pasting.

+

Decision Trees: with random split

==

Bagging with **Decision Trees** as classifier. The trees The trees use a random subset of features at each split

==

An **ensemble learning method** composed only of Decision Trees. They are trained on different random subsets of training data (with or without replacement) . To split a node, the trees select the best features from a random subset of features .

Random forest in sklearn: method 1

- Using a Bagging Classifier: since a random forest is a special case, of a bagging method. We can use a Bagging Classifier with decision trees as estimator.

```python
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
myEstimator= DecisionTreeClassifier(splitter="random",max_features="auto",max_depth=4)
myRF1 = BaggingClassifier(myEstimator,n_estimators=500,
                          max_samples=1.0, bootstrap=True, n_jobs=-1,oob_score=True)

myRF1.fit(x_train,y_train)
y_pred= myRF1.predict(x_test)
print("Confusion matrix = \n",confusion_matrix(y_test,y_pred))
print("score=", np.round(myRF1.score(x_test,y_test),2))
```

```
Confusion matrix =
 [[11  0  0]
 [ 0 10  0]
 [ 0  1 16]]
score= 0.97
```

By specifying the splitter= “random”, at each split, the chosen feature will be the “**best random feature”:** the best feature from a random subset of features.

[By Amina Delali]

Random forest in sklearn: method 2

1- Random Forest

- Using the random forest classifier class: in sklearn, the size of the samples used in the trees is the same as the original data size.

```
from sklearn.ensemble import RandomForestClassifier

myRF2= RandomForestClassifier(n_estimators=500, n_jobs=-1,max_depth=4,oob_score=True)
myRF2.fit(x_train, y_train)
y_pred = myRF2.predict(x_test)

print("Confusion matrix = \n",confusion_matrix(y_test,y_pred))
print("score=", np.round(myRF2.score(x_test,y_test),2))
```

```
Confusion matrix =
 [[11  0  0]
 [ 0 10  0]
 [ 0  1 16]]
score= 0.97
```

Same parameters used in the bagging classifier for the previous example.

There is no “splitter” parameter. Almost all other default parameters are the same. The only one that differs is : max_features which is here set by default to “auto”. The only parameter we had to specify is (to be adequate with the tree of the previous example) is : max_depth

max_features =“auto” means that:
max_features = sqrt(numb_features)

[By Amina Delali]

Concept

- In **Extreme random forests**, the **thresholds** used in the decision trees are **selected randomly**, instead of choosing the best one.

- They are also called "**Extremely randomized Trees**" or "**Extra Trees**".

- In sklearn, they are: "**ExtraTreesClassifier**" and "**ExtraTreesRegeressor**"

- "Extra**Trees**Classifier" are random forests, but there is in sklearn "Extra**Tree**Classifier" which is a decision tree and not a forest.

2- Extreme Random Forest

[By Amina Delali]

2- Extreme Random Forest

Example

```python
from sklearn.tree import ExtraTreeClassifier

myEstimator2= ExtraTreeClassifier(splitter="random",max_features="auto",max_depth=4)
myERF1 = BaggingClassifier(myEstimator2,n_estimators=500,
                           max_samples=1.0, bootstrap=True, n_jobs=-1,oob_score=True)

myERF1.fit(x_train,y_train)
y_pred = myERF1.predict(x_test)

print("Confusion matrix = \n",confusion_matrix(y_test,y_pred))
print("score=", np.round(myERF1.score(x_test,y_test),2))
```

```
Confusion matrix =
 [[11  0  0]
 [ 0 10  0]
 [ 0  1 16]]
score= 0.97
```

```python
from sklearn.ensemble import ExtraTreesClassifier

myERF2= ExtraTreesClassifier(n_estimators=500, n_jobs=-1,max_depth=4,oob_score=True,bootstrap=True)
myERF2.fit(x_train, y_train)
y_pred = myERF2.predict(x_test)

print("Confusion matrix = \n",confusion_matrix(y_test,y_pred))
print("score=", np.round(myERF2.score(x_test,y_test),2))
```


[By Amina Delali]

2- Extreme Random Forest

Out of bag scores

- The bag refers to the training samples selected and used by a predictor in a bagging method.
- The out of bag, are the remaining samples.
- For each predictor, it is possible to compute its score using its out of bag samples.
- At the end of the training, the average of all these scores will represent the "out of bag score" of the learning method.
- In sklearn, you have to specify: "oob_score =True" as parameter of the bagging method.

```
myRF1 = BaggingClassifier(myEstimator,n_estimators=500,
                          max_samples=1.0, bootstrap=True, n_jobs=-1,oob_score=True)
```

```
myRF2= RandomForestClassifier(n_estimators=500, n_jobs=-1,max_depth=4,oob_score=True)
```

```
print("Out of bag score:", np.round(myRF1.oob_score_,2))
print("Out of bag score:", np.round(myRF2.oob_score_,2))
print("Out of bag score:", np.round(myERF1.oob_score_,2))
print("Out of bag score:", np.round(myERF2.oob_score_,2))
```

```
Out of bag score: 0.94
Out of bag score: 0.95
Out of bag score: 0.96
```

3- Balanced Random Forest

Concept

- Sometimes, in classification, the representation of your classes in the training data is "**unbalanced".** For example, in a binary classification, you can have a big number of samples belonging to the first class. And, only few samples related to the other one.

- In this case, the learning method will not be able to classify correctly.

- To avoid this situation, scikit-learn allow us to define a parameter called: **class_weight**, with a special value: "**balanced**" ==> it will attribute bigger weights for less present classes.

- The formula is as follow: n_samples / (n_classes * np.bincount(y))

Count number of occurrences of each value in y

Example without balancing

```
from pandas import DataFrame as DF, Series as S
import pandas as pd
myData = pd.read_csv("AAA-Ped-Week5/A3P-w5-data_imbalance.txt", header = -1)

x2= myData.iloc[:,[0,1]].values
y2= myData.iloc[:,2].values
```

	0	1	2
0	5.66	6.77	1
1	4.40	5.05	0
2	3.52	4.73	1

Loading the data, and extracting **x** and **y** values

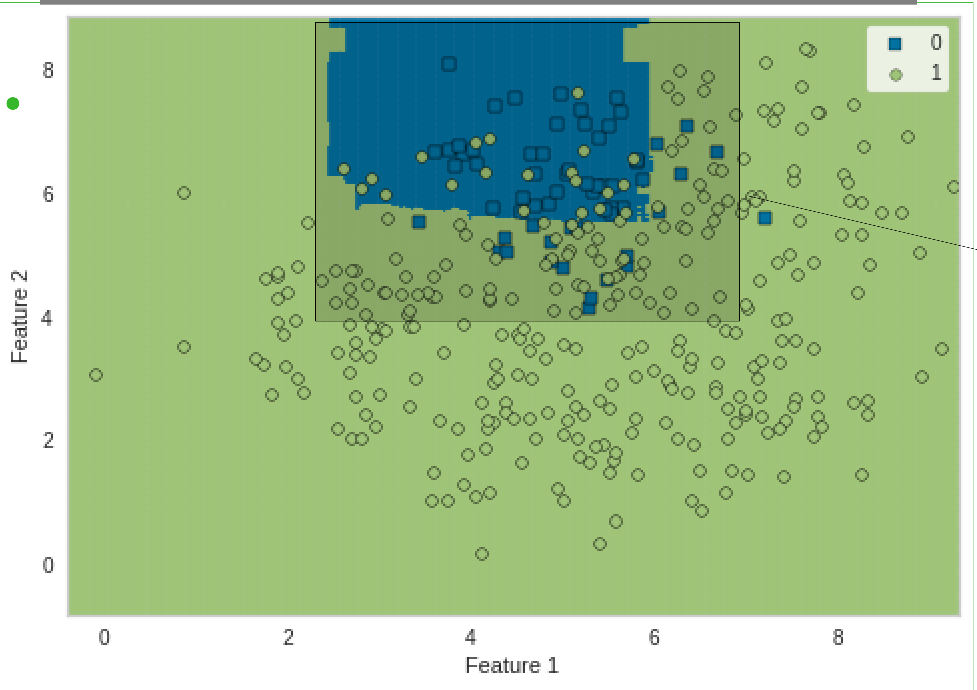
We will use **yellowbrick** library to visualize our classification boundaries.

```
from yellowbrick.contrib.classifier import DecisionViz


myRF2= RandomForestClassifier(n_estimators=100, n_jobs=-1,oob_score=True, max_depth=4,
)
myUVis = DecisionViz(myRF2,"Unbalanced Learning",features = ["Feature 1","Feature 2"],
                     classes = [0,1])


myUVis.fit(x2_train,y2_train)
myUVis.draw(x2_test, y2_test)
myUVis.poof(outpath="unbalanced.png")
```

3- Balanced Random Forest

[By Amina Delali]

Example without balancing (suite)



The classes are not balanced

We see that some samples belonging to class 0 are predicted to be in class 1 and vice versa.

	precision	recall	f1-score	support
0	0.65	0.66	0.66	62
1	0.93	0.93	0.93	313

3- Balanced Random Forest

[By Amina Delali]

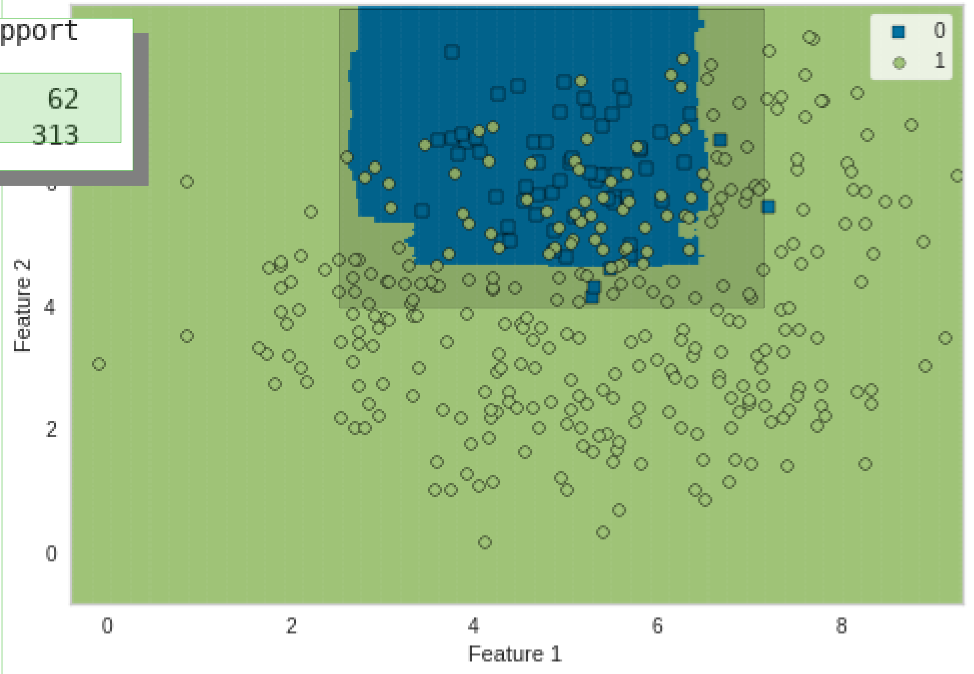
Same example with balancing

```
myBRF2= RandomForestClassifier(n_estimators=100, n_jobs=-1,oob_score=True, max_depth=4,
                               class_weight="balanced")
myVis = DecisionViz(myBRF2,"Balanced Learning",features = ["Feature 1","Feature 2"],
                    classes = [0,1])


myVis.fit(x2_train,y2_train)
myVis.draw(x2_test, y2_test)
myVis.poof(outpath="balanced.png")
```

Balanced classes

	precision	recall	f1-score	support
0	0.53	0.92	0.67	62
1	0.98	0.84	0.91	313

- We see clearly that the decision boundaries changed: the area of class 0 is bigger now
- But, we have more classes 1 predicted as belonging to class 0.



[By Amina Delali]

3- Balanced Random Forest

4- Grid Search

Concept

- **GridSearch** is a tool used to test a learning model with a list of dictionaries containing different parameters.
- We will use it to find the best parameters for a **Random Forest** method. We will apply it on the already used **iris** data.

```
# variate n_estimators, max_depth; and min_samples_leaf
myGrid = [ {'n_estimators': [20,40,60], "min_samples_leaf":[1,2,3]},
           { 'n_estimators': [10,20,30,40], 'max_depth': [1,2,3]}
           ]

from sklearn.model_selection import GridSearchCV
# A grid search instanciation using a Random Forest classifier
myEstimator = RandomForestClassifier()
myClassifier = GridSearchCV(estimator= myEstimator,
     param_grid= myGrid)
myClassifier.fit(x_train, y_train)
```

First the first dictionary we have: 9 combinations of parameters. For the second one we have: 12 combinations.

```
print("\nThe obtained scores are:")
for params, mean, allv in myClassifier.grid_scores_:
  print(params, '-->', np.round(mean, 3))
print("The best parameters are :", myClassifier.best_params_)
```

[By Amina Delali]

4- Grid Search

Example results

```
{'min_samples_leaf': 1, 'n_estimators': 20} ==> 0.964
{'min_samples_leaf': 1, 'n_estimators': 40} ==> 0.964
{'min_samples_leaf': 1, 'n_estimators': 60} ==> 0.964
{'min_samples_leaf': 2, 'n_estimators': 20} ==> 0.973
{'min_samples_leaf': 2, 'n_estimators': 40} ==> 0.973
{'min_samples_leaf': 2, 'n_estimators': 60} ==> 0.964
{'min_samples_leaf': 3, 'n_estimators': 20} ==> 0.973
{'min_samples_leaf': 3, 'n_estimators': 40} ==> 0.973
{'min_samples_leaf': 3, 'n_estimators': 60} ==> 0.964
{'max_depth': 1, 'n_estimators': 10} ==> 0.946
{'max_depth': 1, 'n_estimators': 20} ==> 0.955
{'max_depth': 1, 'n_estimators': 30} ==> 0.964
{'max_depth': 1, 'n_estimators': 40} ==> 0.955
{'max_depth': 2, 'n_estimators': 10} ==> 0.955
{'max_depth': 2, 'n_estimators': 20} ==> 0.964
{'max_depth': 2, 'n_estimators': 30} ==> 0.964
{'max_depth': 2, 'n_estimators': 40} ==> 0.973
{'max_depth': 3, 'n_estimators': 10} ==> 0.964
{'max_depth': 3, 'n_estimators': 20} ==> 0.964
{'max_depth': 3, 'n_estimators': 30} ==> 0.964
{'max_depth': 3, 'n_estimators': 40} ==> 0.964
```

- We can see that augmenting the number of estimators doesn't necessary enhance the results.
- Same observation about max_depth value

```
The best parameters are : {'min_samples_leaf': 2, 'n_estimators': 20}
```

[By Amina Delali]

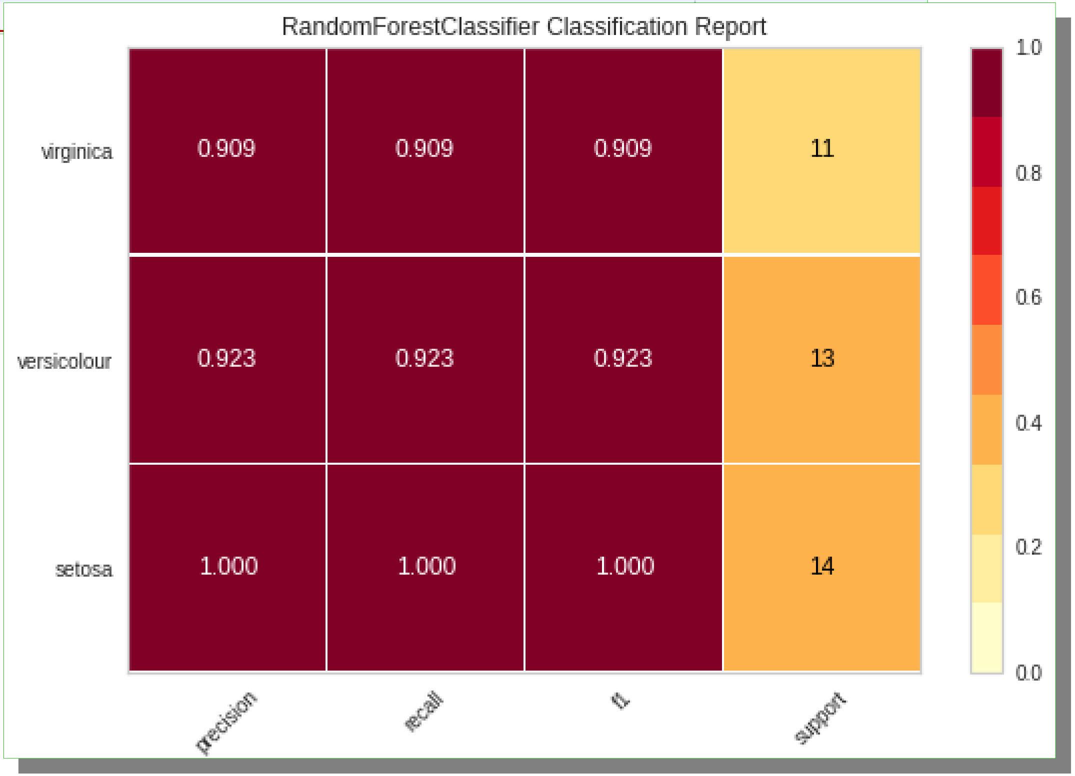
4- Grid Search

Best result visualization

- We will display the confusion matrix using **yellowbrick.**

```
from yellowbrick.classifier import ConfusionMatrix
myEstimator= RandomForestClassifier(min_samples_leaf=2,n_estimators= 20)
myVisualizer = ClassificationReport(myEstimator, classes=["setosa","versicolour","virginica"],
                                    support=True)
myVisualizer.fit(x_train,y_train)
myVisualizer.score(x_test,y_test)
myVisualizer.poof(outpath="Confusion_
```

The confusion matrix is printed as a “heat-map”: the best values are the darkest, and vice-versa.



[By Amina Delali]

5- Feature Importance

Concept

- The **features** in a data set have **not** the **same importance**.
- In a **decision tree**, this importance can be deduced from the **distance** between the **appearance** of a **feature** in a **decision node** and **the root**.
- In a **random forest**, the **average** of the distances corresponding to each tree will represent the **feature's importance**.
- **Scikit-learn** implements this method.

```python
# we will use the previous trained estimator

for feature, importance in zip(myIris["feature_names"], myEstimator.feature_importances_):
  print(feature,'==>' , np.round(importance,5))
```

```
sepal length (cm) ==> 0.07181
sepal width (cm) ==> 0.00342
petal length (cm) ==> 0.3492
petal width (cm) ==> 0.57557
```

We can see clearly that the most important features are: “petal length” and “petal width”

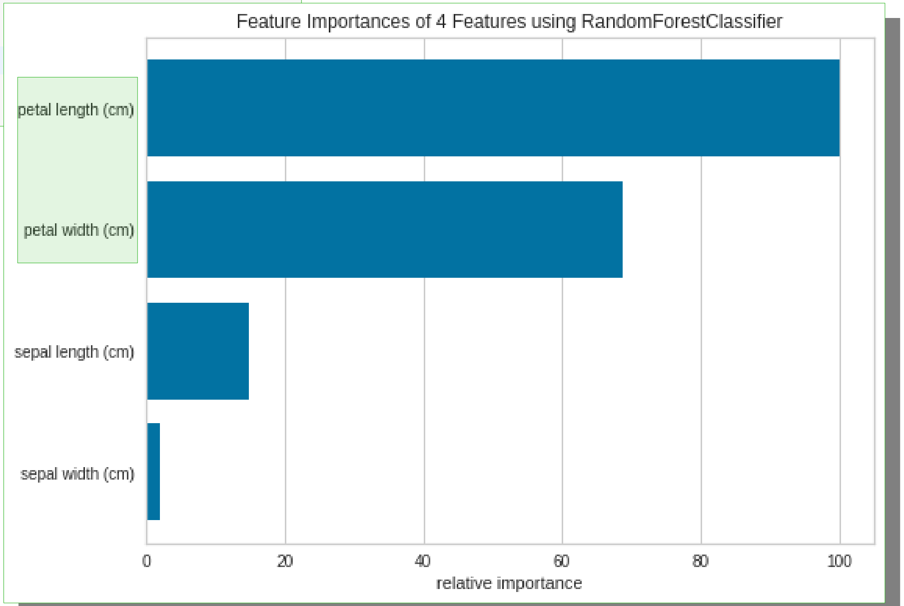
[By Amina Delali]

5- Feature Importance

Visualization of the features importance

- Again, we will use **yellowbrick.**

```
from yellowbrick.features.importances import FeatureImportances

#features names
features = myIris["feature_names"]
# instanciate the feature importance visualizer
FIVisualizer = FeatureImportances(myEstimator, labels=features)
# train the visualizer
FIVisualizer .fit(X, y)
# display the realitive features importance
FIVisualizer.poof("feature_importance.png")

```



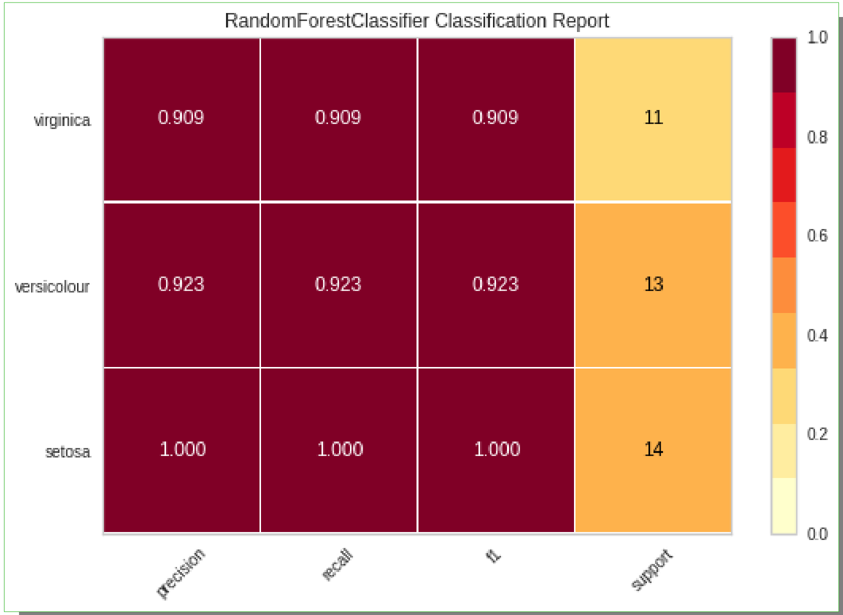
Most important features

[By Amina Delali]

5- Feature Importance

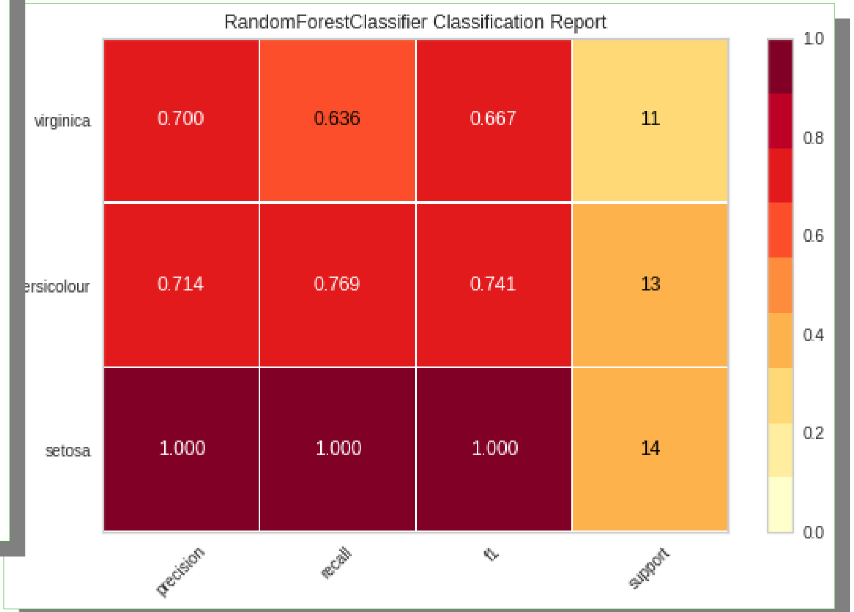
Comparison of results

```
myVisualizer.fit(x_train[:,[2,3]].reshape(-1,2),y_train)
```



Using only **2 most important** features (same result as using all the features)

```
myVisualizer.fit(x_train[:,[0,1]].reshape(-1,2),y_train)
```



Using only **2 least important** features (same result as using all the features)

[By Amina Delali]

6- Balancing by resampling

Concept

- An other way to obtain a balanced training set, is to **remove** samples from the majority class. Which is called: **under-sampling.** Or, to **add** new samples belonging to the less represented class. Which is called: **over-sampling.**

- There is a variety of resampling techniques. The library **imbalanced-learn** library implements some of them.

- For a **RandomForest classifier,** it applies: random under-sampling technique with different strategies.

- The default one is to : resample all classes but the minority class;

Example

6- Balancing by resampling

- In this example, we combine the **2** libraries: **yellowbrick** and **imbalanced-learn**

```
from imblearn.ensemble import BalancedRandomForestClassifier
from sklearn.metrics import balanced_accuracy_score

myBRF3 = BalancedRandomForestClassifier(n_estimators=100, n_jobs=-1,oob_score=True, max_depth=4)
myVis3 = DecisionViz(myBRF3,"Balanced Learning2",features = ["Feature 1","Feature 2"],
                    classes = [0,1])


myVis3.fit(x2_train,y2_train)
myVis3.draw(x2_test, y2_test)
myVis3.poof(outpath="balanced2.png")
```

```
print(classification_report(y2_test,myBRF3.predict(x2_test)))
```

	precision	recall	f1-score	support
0	0.50	0.89	0.64	72
1	0.97	0.79	0.87	303
micro avg	0.81	0.81	0.81	375
macro avg	0.73	0.84	0.75	375
weighted avg	0.88	0.81	0.82	375

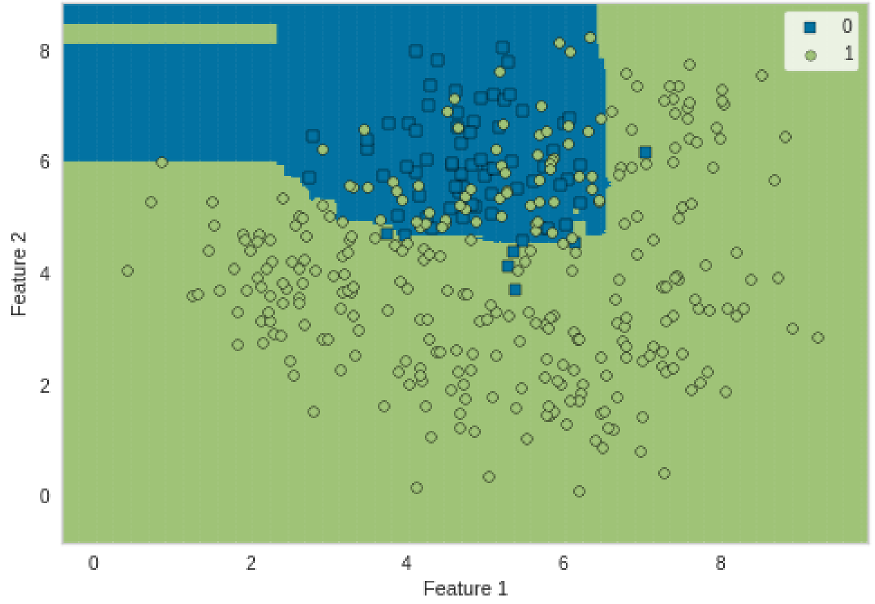
[By Amina Delali]

Example (suite)

6- Balancing
by resampling

```python
from IPython.display import Image
Image(filename='balanced2.png')
```

We can see that the **class 0** region obtained is different than the one obtained using sckit-learn weighting technique



[By Amina Delali]

References

- Aurélien Géron. Hands-on machine learning with Scikit-Learn and Tensor-Flow: concepts, tools, and techniques to build intelligent systems. O’Reilly Media, Inc, 2017.
- imbalanced learn. User guide. On-line at https://imbalanced-learn.org/en/stable/user_guide.html. Accessed on 09-12-2018.
- Joshi Prateek. Artificial intelligence with Python. Packt Publishing, 2017.
- Alencar Rafael. Resampling strategies for imbalanced datasets. On-line at https://www.kaggle.com/rafjaa/resampling-strategies-for-imbalanced-datasets Accessed on 09-12-2018.
- Scikit-learn.org. scikit-learn, machine learning in python. On-line at https://scikit-learn.org/stable/. Accessed on 03-11-2018.
- yellowbrick. Yellowbrick: Machine learning visualization. On-line at http://www.scikit-yb.org/en/latest/. Accessed on 09-12-2018.

Thank you!

FOR ALL YOUR TIME