Birkbeck, University of London

Machine Learning - COIY065H7 Coursework Report

Baran Buluttekin
13153116

bbulut02@dcs.bbk.ac.uk

I have read and understood the sections of plagiarism in the College Policy on assessment offences and confirm that the work is my own, with the work of others clearly acknowledged. I give my permission to submit my report to the plagiarism testing database that the College is using and test it using plagiarism detection software, search engines or meta-searching software.

Abstract

This report is the summary for my findings in classification of the yeast dataset [9]. In the introduction general information is given about the dataset and challenges associated with this problem is laid out. Section 2 gives an background information about techniques and algorithms that will be used to solve the problem. Methodology section gives more detailed breakdown of the steps for solution. Experiments and results section will introduce metrics and summary statistics obtained from the experiments. Conclusion section will highlight general findings and offer a next steps for the improvement.

There is also appendix section at the end that includes python code used to carry out the experimentations. Even though it will produce the results of the experiments it will take a while to run due to the high number of experimentations and parameter tuning used. Therefore please consider using jupyter notebook provided with the submission to reproduction.

Contents

1 Introduction

This report is part of the coursework for *Machine Learning* (COIY065H7) module. From the coursework instructs Yeast data [9] has chosen as the dataset.

1.1 Dataset

Yeast dataset is part of the UCI machine learning repository where variety of machine learning related datasets curated. This dataset includes properties of yeast proteins to predict localization site of the protein. Number of data points in the dataset is 1484 and have 9 features in which only 8 of them are predictive feature. First feature is only index indicating database of that data acquired and have no predictive purpose. Final column of the data set provides label for protein class that we are aiming to predict. This dataset have no missing values. Given the labelled nature of the data this classification problem can be solved using *supervised learning*.

1.1.1 Problem Statement

There are some additional properties of this data makes classification task challenging. Firstly this is a multi class classification problem, because our dataset have 10 distinct labels. Labels extracted from dataset are, 'MIT', 'NUC', 'CYT', 'ME1', 'EXC', 'ME2', 'ME3', 'VAC', 'POX' and 'ERL'. Secondary challenge is that this dataset have class imbalance. Number of instances varies form 463 observation of 'CYT' to 5 instances of 'ERL'.

2 Algorithms and Techniques

There are number of techniques and algorithms introduced to address the challenges mentioned in the subsection 1.1.1. I will list some of this techniques below and explain more detailed in the fallowing sections.

- **Over-sampling data:** This method oversample the minority class in the data to create balanced classes.
- **Under-sampling data:** Opposite of the item above this method under sample the majority class to balance the data.
- **Balanced sampling for ensemble:** Sampling data in a balanced way while building ensemble models.
- **Choosing performance matrix sensitive to imbalance:** Using evaluation metric such as f1 score instead of accuracy will better capture mis-classification errors.

Most of the techniques discussed above are implemented as a code package in imbalance-learn [6]. In the case of over-sampling, only random sampling method available for multi class

datasets. Choice of algorithm for under-sampling is substantially more than over-sampling. But due to the very large gap in the number of instances in the different classes, under-sampling can effect performance of the classification model significantly. For example if we were to under-sample until we reached to balance dataset because of the only 5 instance in the class 'ERL' we would get a dataset of 50 instances (10 class times 5 observation each). Regardless I will be using alternative under-sampling algorithm design to experiment their performance.

2.1 Under-sampling methods

In addition to random under-sampling method special algorithms to under-sample selectively will be used in experiments. One such algorithm is *Condensed nearest neighbour* (CNN) [3], utilizes one nearest neighbor rule to decide for each instance to be removed from the dataset or not. Algorithm steps from the article are [3]:

> "Get all minority samples in a set C.
>
> Add a sample from the targeted class (class to be under-sampled) in C and all other samples of this class in a set S.
>
> Go through the set S, sample by sample, and classify each sample using a 1 nearest neighbor rule.
>
> If the sample is misclassified, add it to C, otherwise do nothing.
>
> Reiterate on S until there is no samples to be added."

Neighbor cleaning rule (NCL) [5] is another algorithm that derived from CNN where more emphasis give to the data cleaning.

2.2 Sampling for ensemble

Ensemble models uses subset of the data with replacement while training different classifier but this sapling usually does not take class imbalance into account. Classifiers in the imbalance-learn library build in a way that sampling is done while maintaining the class balance. Below is the short description of these classifiers.

Balanced random forest [2] classifier is design such a way that each bootstrap sample to train tree in the forest is supplied with balanced class subset. This algorithm uses classic random forest [1] implementation and only differs in the sampling methodology.

RUSBoost [12] also randomly sample data in balanced way before the boosting implemented.

Easy Ensemble [7] is an AdaBoost [11] implementation trained on balanced bootstrap samples.

3 Methodology

As an initial step, I have loaded the dataset in a data frame using Pandas [8] package which then allow for exploratory examination of the data. For that reason I have created a pairplot to observe if there is any linear separation is possible. Please fallow the steps of the code snippet below to reproduce the visualization. All variable in the data is paired to each other except the "pox" and "erl" which behaves more like categorical data then continuous data.

Base line machine algorithms like random forest, AdaBoost, Support vector classifier and others are part of Scikit-learn [10] package. All visualization created are achieved using Matplotlib [4] and Seaborn [13] software packages. For reproducibility random number state is chosen and in variable named as RANDOM_STATE. All non-deterministic algorithm uses this random generator seed.

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

url = "https://archive.ics.uci.edu/ml/machine-learning-databases/yeast/
                                        yeast.data"
columns = ["mcg", "gvh", "alm", "mit", "erl", "pox", "vac", "nuc", "class"
                                        ]
df = pd.read_csv(url, header=None, sep=r"\s+", names=columns, usecols=list
                                        (range(1,10)))

feature_list = [x for x in df.columns if x not in ['pox', 'erl']]
# Plotting
sns.pairplot(df[feature_list], hue='class')
```

Figure 1: Pairplot of the data.

3.1 Data preprocessing

Data preprocessing is relatively simple for this dataset. There are no missing values and all data points are numeric which only leaves us to encode class variable to be able to feed into machine learning model. Judging from pairplot above classes have different range in some variable and can benefit min max scaling of the data.

In later stages some regularization algorithms will be applied to data to observe its effects in the generalization.

This preparation can be streamlined with fallowing code.

```
def process_data(df):
    encoder = LabelEncoder()
    y = encoder.fit_transform(df['class'])
    X = df[df.columns[:-1]].values
    return X, y

def process_scaled_data(df):
    l_encoder = LabelEncoder()
    scaler = MinMaxScaler()
    y = l_encoder.fit_transform(df['class'])
    X = df[df.columns[:-1]].values
    X = scaler.fit_transform(X)
    return X, y
```

3.2 Over-sampling and training model

Defined over-sampling method will be applied and compare the result with the same classifier trained on the data to observe if over-sampling help the classification task. Similarly utility function is created for the ease of transformation.

```
def make_over_sample(X, y, random_state=0):
    ros = RandomOverSampler(random_state=random_state)
    return ros.fit_resample(X, y)
```

3.3 Under-sample and train model

I previously discuss that we have different algorithm for under-sampling and details for these algorithms given in the subsection 2.1. These algorithms implemented in imbalance-learn package as *CondensedNearestNeighbour*, *NeighbourhoodCleaningRule* and *RandomUnderSampler*. Similarly these will be compared to base models trained on the data to evaluate if under-sampling helps the classification. Also implemented with utility function.

```
def make_under_sample(X, y, method=NeighbourhoodCleaningRule,
                                        random_state=0):
    clf = method(random_state=random_state)
    return clf.fit_resample(X, y)
```

3.4 Comparison of models

All of the experimentations will be carried out by firstly splitting data to training and testing data. For the splitting *StratifiedKFold* algorithm is chosen because of the minority class in the dataset. If the random splitting algorithm is used minority class can either not represented in test data or training data. StratifiedKFold will ensure each class is represented in the training and test dataset. All machine learning algorithm and sampling techniques will be applied to training data after the split so the effect of each process can be examined on

unseen data when it tested with test data. This rule also applies to any scaling on the data. Applying scaling separately to test and training data especially important because applying scaling before splitting will cause information leakage and will cause to misleading results in testing.

Main choice of evaluation metric for this classification is f1 score because of its nature of combining precision and recall. F1 score calculates as harmonic mean of precision and recall, calculated with formula:

$$H = \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \cdots + \frac{1}{x_n}} = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$$

$$F_1 = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} = 2 \times \frac{precision \times recall}{precision + recall}$$

Harmonic mean differ from arithmetic mean in a way that it will give more weight to low value, hence f1 score will increase only if both precision and recall increases.

For each experimentation I will use Random forest, AdaBoost, logistic regression, and svm classifier to observe and compare performance of each method.

Random forest classifier is an ensemble of decision trees that uses the bagging method. Data points and the features are sampled from data to create diverse set of trees. AdaBoost is also an ensemble model that uses bootstrap bagging but mainly differ by assigning a weigh and then emphasising an error in prediction with these weight to build an bagging model. Both of these algorithms will be tuned using number of trees which is implemented in the scikit-learn as "n_estimators".

Logistic regression classifier is similar to logistic regression which fits a mathematical model to underlie data and optimizing parameters θ together with a bias term θ_0.

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$
$$\hat{y} = \theta^T \cdot x$$

Then this fitter equation passed to *logit* function to make a prediction between 2 class (0 and 1)

$$\sigma(z) = \frac{1}{1 + e^{-z}} = \frac{1}{1 + e^{-\theta^T \cdot x}}$$

Despite being originally design as binary classification algorithm it can be applied to multi class classification as well. In scikit-learn this can be achieved by using multi_class="multinomial" option in the instance.

In general first over-sampling and under-sampling method will be compared to classification models build on original data. Then another comparison will be drawn on balanced ensemble models versus general ensemble models which later all will be compared. This process will be repeated on the scaled data to observe scaling effect. All models will be trained with hyper-parameter tuning using cross validation.

4 Experiments and Results

4.1 Ensemble models performance

Selected ensemble models first trained with different hyper-parameters and best performing parameter selected to fit the data using number of estimators in all of the classifiers. Table below summarize the maximum f1 score for each classifier in training and test data. Furthermore all ensemble models trained in [100, 200, 300, 400, 500] numbers of trees and their f1 score observed as well as plotted to below graph. Performance of the algorithm on test set drawn using dotted line and test set score drawn in solid line using same color.

Classifier	F1 score (training)	F1 score (test)
RandomForestClassifier	**1.0**	**0.6686**
AdaBoostClassifier	0.3670	0.3892
EasyEnsembleClassifier	0.5001	0.5917
BalancedRandomForestClassifier	0.4890	0.5148
BalancedBaggingClassifier	0.5633	0.6035

If we observe the f1 score of all the classifiers in the all n_estimators we can see after the random forest best performing imbalanced ensemble model was BalancedBaggingClassifier.

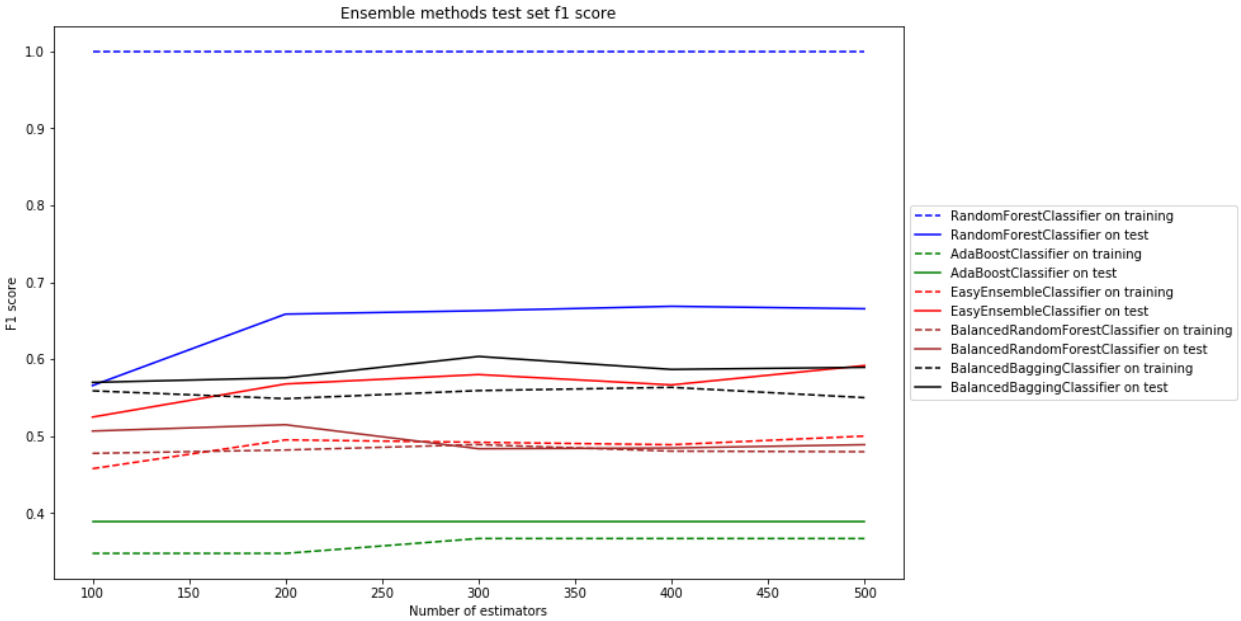

Figure 2: F1 score of ensemble models.

We can investigate RandomForestClassifier predictions further by visualizing its confusion matrix. Which show each number of correctly classified prediction in each class.

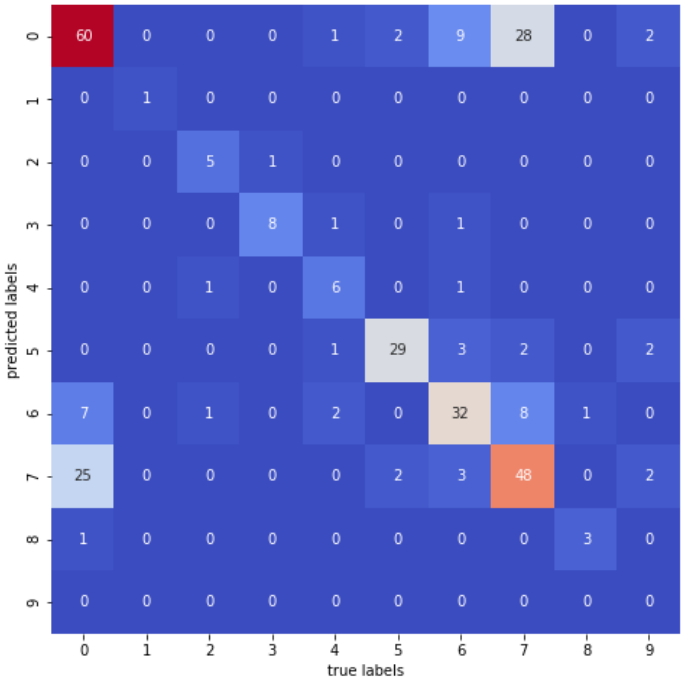

Figure 3: Confusion matrix of RandomForestClassifier.

We can see that "CYT" (0) class is often mis-classified as "NUC" (7) and vice versa. Rest of the classes predicted well.

4.2 Over-sampling v baseline classifiers

As a first step over sampled dataset made available with the random over-sampling defined in subsection 3.2, number of class instances in the over-sampled data. Presented below.

Class	Over-sampled instances	Training dataset instances
0	370	370
1	370	4
2	370	28
3	370	35
4	370	40
5	370	130
6	370	195
7	370	343
8	370	16
9	370	24

Similar to previous subsection I trained ensemble models using number of tree parameter "n_estimators" from 100 to 1000 incrementing 100 in each training. Same algorithm first train in the training data and them trained on over-sampled dataset. F1 scores are tested in training dataset and test data. Below is the change of the performance metric over the number of trees in the algorithm. Algorithm trained on the over-sample data denoted with "_o" at the end of algorithm such as "RandomForestClassifier_o".

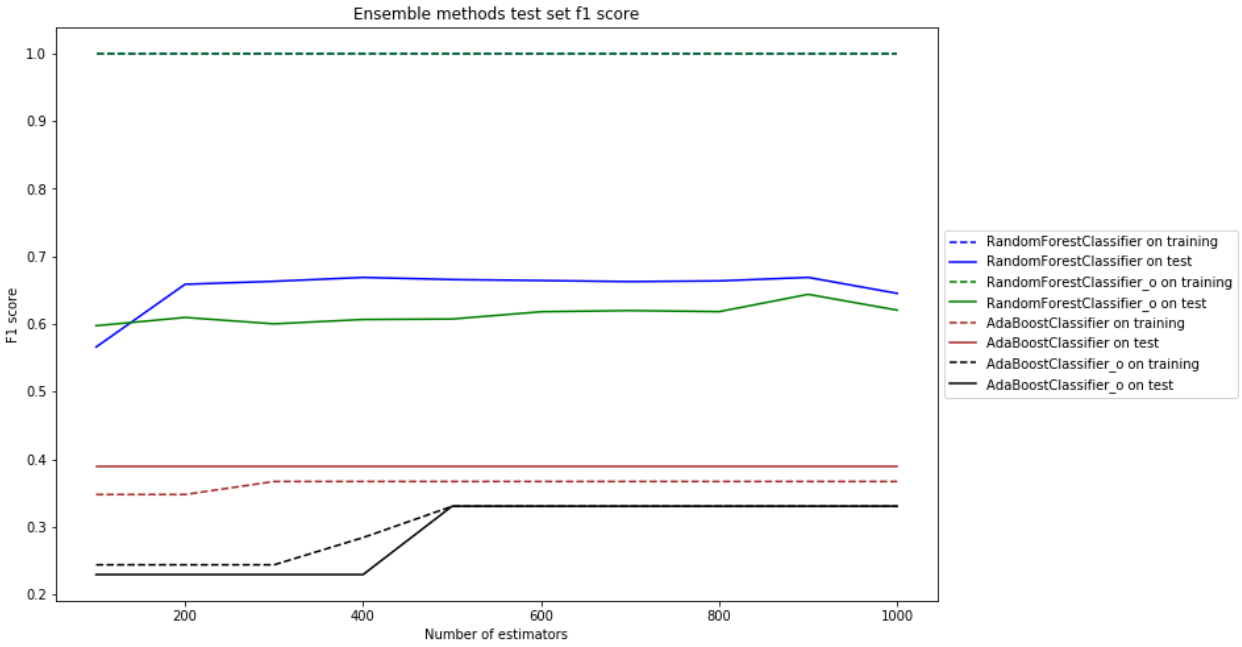

Figure 4: Over-sampled v base classifiers.

Both random forest and the AdaBoost performed slightly better when they trained base dataset.

Logistic regression and support vector classifier (SVC) also trined in the both dataset to check if over-sampled data have different effect on algorithms that are not ensemble

algorithms. Both of these algorithms performed slightly better in the over-sampled data than base dataset. Summary score for all the algorithms presented below.

F1 Scores				
Algorithm Name	Base Dataset		Over-sampled	
	Training	Test	Training	Test
RandomForest	1.0	0.6686	1.0	0.6436
AdaBoost	0.3670	0.3892	0.3302	0.3305
SVC	0.5923	0.5981	0.5980	0.6069
LogisticRegression	0.5847	0.5542	0.5105	0.5568

4.3 Under-sampling v baseline classifiers

Experimentation from previous section repeated for all the algorithms used in the previous section for the under sampling. Algorithms for under-sampling discussed in subsection 3.3 and the dataset instances produced by using these presented below.

Class	Dataset	RUS	CNN	NCL
0	370	4	2	243
1	4	4	4	4
2	28	4	1	17
3	35	4	1	25
4	40	4	2	10
5	130	4	2	86
6	195	4	2	95
7	343	4	4	162
8	16	4	2	7
9	24	4	2	0

For the case of under-sampling algorithms trined on original training set out performed under-sampled peers overwhelmingly for random under-sampling (RUS) and Condensed-NearestNeighbour (CNN). NeighbourhoodCleaningRule (NCL) on the other hand performed slightly worse than baseline. These results were expected for the under-sampling case because of the reason that very few data points left in the dataset for most of the under-sampling technique. Full performance comparison provided in the table below.

F1 Scores								
Algorithm Name	Base Dataset		RUS		CNN		NCL	
	Training	Test	Training	Test	Training	Test	Training	Test
RandomForest	1.0	0.6686	0.3946	0.4287	0.2708	0.2509	0.6366	0.6362
AdaBoost	0.3670	0.3892	0.1330	0.1807	0.05153	0.0446	0.2903	0.3160
SVC	0.5923	0.5981	0.3356	0.3830	0.1450	0.1447	0.5619	0.5887
LogisticRegression	0.5847	0.5542	0.3276	0.3978	0.3291	0.3187	0.6023	0.5937

4.4 Using regularization

As a final comparison dataset is scaled with min max regularizer from scikit-learn [10] package. Ensemble classifiers trained on base dataset and scaled dataset to compare the effect of scaling in the data. Performance of the scaled data was considerably lower than the classifier trained on original data. One reason for this performance reduction can be relative power of the data has a fundamental importance as we can see in the confusion matrix that common mis-classification between classes increases when scaling applied to the dataset.

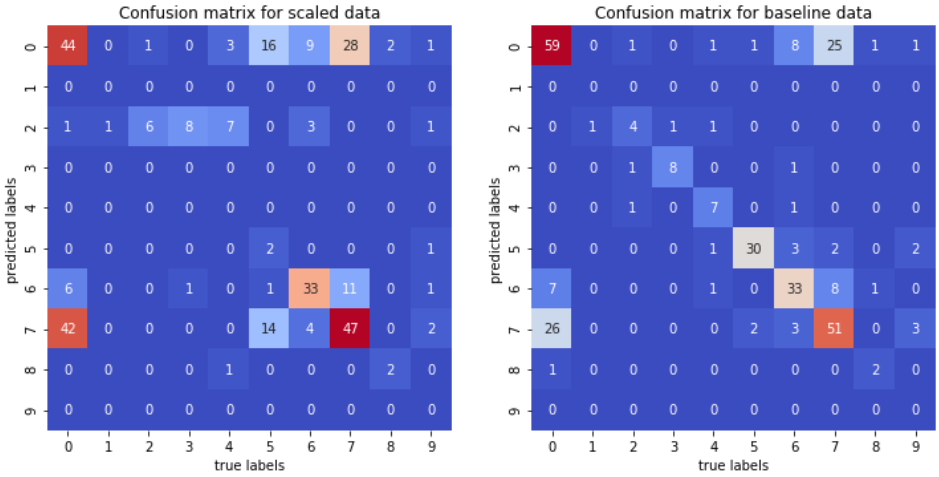

Figure 5: Comparison of same Random forest model in scaled and baseline data.

5 Conclusion

In conclusion, common imbalance techniques had very limited impact on this classification problem. Under-sampling algorithms was not successful mostly because of large number of data loss when the technique applied. Over-sampling and balanced sample bootstrapped ensemble algorithms had similar performance to predictions those without. Albeit slightly less successfully. In general combination of very steep difference in imbalance of the classes and having moderately small data kept this classification problem still a challenging task.

Further improvements can be achieve by training different models or trying additional techniques such as stacking to gain more accuracy.

References

[1] Leo Breiman. "Random Forests". In: *Mach. Learn.* 45.1 (Oct. 2001), pp. 5–32. ISSN: 0885-6125. DOI: 10.1023/A:1010933404324. URL: https://doi.org/10.1023/A:1010933404324.

[2] Chao Chen. "Using Random Forest to Learn Imbalanced Data". In: 2004.

[3] P. Hart. "The Condensed Nearest Neighbor Rule (Corresp.)" In: *IEEE Trans. Inf. Theor.* 14.3 (Sept. 2006), pp. 515–516. ISSN: 0018-9448. DOI: 10.1109/TIT.1968.1054155. URL: https://doi.org/10.1109/TIT.1968.1054155.

[4] J. D. Hunter. "Matplotlib: A 2D Graphics Environment". In: *Computing in Science Engineering* 9.3 (May 2007), pp. 90–95. ISSN: 1521-9615. DOI: 10.1109/MCSE.2007.55.

[5] Jorma Laurikkala. "Improving Identification of Difficult Small Classes by Balancing Class Distribution". In: *Proceedings of the 8th Conference on AI in Medicine in Europe: Artificial Intelligence Medicine*. AIME '01. Berlin, Heidelberg: Springer-Verlag, 2001, pp. 63–66. ISBN: 3-540-42294-3. URL: http://dl.acm.org/citation.cfm?id=648155.757340.

[6] Guillaume Lemaître, Fernando Nogueira, and Christos K. Aridas. "Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning". In: *Journal of Machine Learning Research* 18.17 (2017), pp. 1–5. URL: http://jmlr.org/papers/v18/16-365.

[7] Xu-Ying Liu, Jianxin Wu, and Zhi-Hua Zhou. "Exploratory Undersampling for Class-imbalance Learning". In: *Trans. Sys. Man Cyber. Part B* 39.2 (Apr. 2009), pp. 539–550. ISSN: 1083-4419. DOI: 10.1109/TSMCB.2008.2007853. URL: http://dx.doi.org/10.1109/TSMCB.2008.2007853.

[8] Wes McKinney. "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 51–56.

[9] Kenta Nakai and Paul Horton. *UCI Machine Learning Repository*. Sept. 1996. URL: http://archive.ics.uci.edu/ml/datasets/yeast.

[10] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[11] Robert E. Schapire. "A Brief Introduction to Boosting". In: *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2*. IJCAI'99. Stockholm, Sweden: Morgan Kaufmann Publishers Inc., 1999, pp. 1401–1406. URL: http://dl.acm.org/citation.cfm?id=1624312.1624417.

[12] C. Seiffert et al. "RUSBoost: A Hybrid Approach to Alleviating Class Imbalance". In: *Trans. Sys. Man Cyber. Part A* 40.1 (Jan. 2010), pp. 185–197. ISSN: 1083-4427. DOI: 10.1109/TSMCA.2009.2029559. URL: https://doi.org/10.1109/TSMCA.2009.2029559.

[13] Michael Waskom et al. *mwaskom/seaborn: v0.8.1 (September 2017)*. Sept. 2017. DOI: 10.5281/zenodo.883859. URL: https://doi.org/10.5281/zenodo.883859.

Appendix

```
# -*- coding: utf-8 -*-
"""
*** WARNING ***
This file will take too long to run, if you choose to reproduce
    pls consider using notebook file.

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1
        nyKIP0qveVZwnr5SPnHqAbb2FPfzolfi
"""

import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib import rcParams
from collections import Counter
from sklearn.ensemble import RandomForestClassifier,
    AdaBoostClassifier, ExtraTreesClassifier
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder,
    LabelEncoder, MinMaxScaler, StandardScaler, MaxAbsScaler
from sklearn.model_selection import StratifiedKFold, GridSearchCV,
     cross_val_score, cross_val_predict, train_test_split
from sklearn.svm import LinearSVC, SVC
from sklearn.metrics import auc, confusion_matrix, f1_score,
    precision_score, recall_score, roc_auc_score, roc_curve,
    make_scorer
from sklearn.base import clone
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
```

```
from imblearn.pipeline import make_pipeline
from imblearn.under_sampling import CondensedNearestNeighbour,
    NeighbourhoodCleaningRule, RandomUnderSampler
from imblearn.metrics import classification_report_imbalanced
from imblearn.over_sampling import RandomOverSampler
from imblearn.ensemble import BalancedRandomForestClassifier,
    BalancedBaggingClassifier, EasyEnsembleClassifier
import warnings
warnings.filterwarnings(action='ignore')
from google.colab import files
# %matplotlib inline

rcParams['figure.figsize'] = 12,8

RANDOM_STATE = 123

url = "https://archive.ics.uci.edu/ml/machine-learning-databases/
    yeast/yeast.data"
columns = ["mcg", "gvh", "alm", "mit", "erl", "pox", "vac", "nuc",
    "class"]

df = pd.read_csv(url, header=None, sep=r"\s+", names=columns,
    usecols=list(range(1,10)))
df.head()

def process_data(df):
  encoder = LabelEncoder()
  y = encoder.fit_transform(df['class'])
  X = df[df.columns[:-1]].values
  return X, y

def process_scaled_data(df):
  l_encoder = LabelEncoder()
  scaler = MinMaxScaler()
  y = l_encoder.fit_transform(df['class'])
  X = df[df.columns[:-1]].values
  X = scaler.fit_transform(X)
  return X, y

X, y = process_data(df)
```

```
def make_over_sample(X, y, random_state=0):
  ros = RandomOverSampler(random_state=random_state)
  return ros.fit_resample(X, y)

def make_under_sample(X, y, method=NeighbourhoodCleaningRule,
   random_state=0):
  clf = method(random_state=random_state)
  return clf.fit_resample(X, y)

X_u, y_u = make_under_sample(X, y, random_state=RANDOM_STATE)
Counter(y_u).most_common()

"""## Classifiers

- Random Forest
- Decision tree clf
- logistic regression
- svc
- AdaBoost
"""

rf_params = {
    'bootstrap': [True],
    'max_depth': [80, 90, 100, 110],
    'max_features': [2, 3],
    'min_samples_leaf': [3, 4, 5],
    'min_samples_split': [8, 10, 12],
    'n_estimators': [100, 200, 300, 1000]
}

ros = RandomOverSampler(random_state=RANDOM_STATE)

rf_clf = RandomForestClassifier(random_state=RANDOM_STATE)
rf_bal = BalancedRandomForestClassifier(random_state=RANDOM_STATE)

skf = StratifiedKFold(n_splits=5, random_state=RANDOM_STATE)
clf1_m = {"fpr":[], "tpr":[], "th":[], "mean_fpr":None, "mean_tpr"
   :None}
clf2_m = {"fpr":[], "tpr":[], "th":[], "mean_fpr":None, "mean_tpr"
   :None}
```

```
for train_index, test_index in skf.split(X, y):
  X_train = X[train_index]
  y_train = y[train_index]
  X_test = X[test_index]
  y_test = y[test_index]
  clone_clf = clone(rf_clf)
  clone_bal = clone(rf_bal)
  #scorer = make_scorer(loss_func, greater_is_better=True)
  grid_clf = GridSearchCV(estimator=clone_clf, param_grid=
     rf_params, cv=5, verbose=2, scoring="f1_macro")
  grid_bal = GridSearchCV(estimator=clone_bal, param_grid=
     rf_params, cv=5, verbose=2, scoring="f1_macro")
  grid_clf.fit(X_train, y_train)
  print("base classifier trained")
  grid_bal.fit(X_train, y_train)
  print(f1_score(y_test, grid_clf.predict(X_test), average="macro"
     ))
  print(f1_score(y_test, grid_bal.predict(X_test), average="macro"
     ))
  break

skf = StratifiedKFold(n_splits=5, random_state=RANDOM_STATE)
clf1_m = {"fpr":[], "tpr":[], "th":[], "mean_fpr":None, "mean_tpr"
   :None}
clf2_m = {"fpr":[], "tpr":[], "th":[], "mean_fpr":None, "mean_tpr"
   :None}

for train_index, test_index in skf.split(X, y):
  X_train = X[train_index]
  y_train = y[train_index]
  X_test = X[test_index]
  y_test = y[test_index]
  break

dt_grid = {'max_depth': np.arange(3, 10)}

tree = GridSearchCV(DecisionTreeClassifier(random_state=
   RANDOM_STATE), dt_grid, cv=5, verbose=1, scoring="f1_macro")

tree.fit(X_train, y_train)

tree.best_params_
```

```
ada_params = {"n_estimators":[10, 50, 100, 500, 1000, 5000],
         "learning_rate":[0.1, 0.5, 0.75, 1.],
         "algorithm":['SAMME', 'SAMME.R']
         }

ada_clf = GridSearchCV(AdaBoostClassifier(random_state=
   RANDOM_STATE), param_grid=ada_params, cv=5, scoring="f1_macro")

ada_clf.fit(X_train, y_train)

ada_clf.best_params_

ada_clf.best_score_

f1_score(y_test, ada_clf.predict(X_test), average="macro")

easy_params = {"n_estimators":[10, 50, 100, 500, 1000, 5000]
         }

easy_clf = GridSearchCV(EasyEnsembleClassifier(random_state=
   RANDOM_STATE, n_jobs=-1), param_grid=easy_params, verbose=2, cv
   =5, scoring="f1_macro")

easy_clf.fit(X_train, y_train)

easy_clf.best_params_

easy_clf.cv_results_

f1_score(y_test, easy_clf.predict(X_test), average="macro")

def comparison_ensemble(X, y, X_test, y_test, n_estimators, clfs,
   random_state=0):
  test_metrics = {}
  train_metrics = {}
  for i, clf in enumerate(clfs):
    for n in n_estimators:
      c = clf(n_estimators=n, random_state=random_state)
      c.fit(X, y)
      print(clf, n)
      y_train_pred = c.predict(X)
```

```
        y_test_pred = c.predict(X_test)
        if n == n_estimators[0]:
          train_metrics[clfs[i]] = [f1_score(y, y_train_pred,
              average="macro")]
        else:
          train_metrics[clfs[i]].append(f1_score(y, y_train_pred,
              average="macro"))
        if n == n_estimators[0]:
          test_metrics[clfs[i]] = [f1_score(y_test, y_test_pred,
              average="macro")]
        else:
          test_metrics[clfs[i]].append(f1_score(y_test, y_test_pred,
               average="macro"))
    return test_metrics, train_metrics

tm, trm = comparison_ensemble(X_train, y_train, X_test, y_test,
    list(range(500, 6000, 500)), [RandomForestClassifier,
    AdaBoostClassifier, EasyEnsembleClassifier,
    BalancedRandomForestClassifier, BalancedBaggingClassifier],
    RANDOM_STATE)

fig = plt.figure()
ax = plt.subplot(111)

for i, j in zip(trm, tm):
    ax.plot(list(range(500, 6000, 500)), tm[i], label="{} training".
        format(str(i).split(".")[-1][:-2]), linestyle="--")
    ax.plot(list(range(500, 6000, 500)), tm[j], label="{} test".
        format(str(j).split(".")[-1][:-2]))
ax.legend()
ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))

newtrm, newtm = comparison_ensemble(X_train, y_train, X_test,
    y_test, list(range(100, 501, 100)), [RandomForestClassifier,
    AdaBoostClassifier, EasyEnsembleClassifier,
    BalancedRandomForestClassifier, BalancedBaggingClassifier],
    RANDOM_STATE)

newtrm

fig = plt.figure(figsize=(12, 8))
ax = plt.subplot(111)
```

```
colors = ['blue', 'green', "red", "brown", "black"]

for i, j, c in zip(newtrm, newtm, colors):
  ax.plot(list(range(100, 501, 100)), newtm[j], label="{} on
     training".format(str(j).split(".")[-1][:-2]), linestyle="--",
      color=c)
  ax.plot(list(range(100, 501, 100)), newtrm[i], label="{} on test
     ".format(str(i).split(".")[-1][:-2]), color=c)

ax.legend()
lgd = ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))
ax.set_ylabel("F1 score")
ax.set_xlabel("Number of estimators")
ax.set_title("Ensemble methods test set f1 score")
fig.savefig("ensembletestf1.png", bbox_extra_artists=(lgd,),
   bbox_inches='tight')
#files.download("ensembletestf1.png")

for i, j in zip(newtm, newtrm):
  print(i, max(newtm[i]), "train")
  print(j, max(newtrm[j]), "test")

rclf = RandomForestClassifier(n_estimators=500, random_state=
   RANDOMSTATE)

rclf.fit(X_train, y_train)

cm = confusion_matrix(y_test, rclf.predict(X_test))
hm = sns.heatmap(cm.T, square=True, annot=True, fmt="d", cbar=
   False, cmap='coolwarm')
plt.xlabel("true labels")
plt.ylabel("predicted labels")
hm.get_figure().savefig("rfheatmap.png")
files.download("rfheatmap.png")

"""## Over-samppling"""

def comparison_oversample(X, y, X_test, y_test, n_estimators, clfs
   , random_state=0):
  test_metrics = {}
  train_metrics = {}
  X_o, y_o = make_over_sample(X, y, random_state=RANDOMSTATE)
```

```
  for i, clf in enumerate(clfs):
    for n in n_estimators:
      c = clf(n_estimators=n, random_state=random_state)
      c.fit(X, y)
      print(str(clf).split(".")[-1][:-2], n)
      y_train_pred = c.predict(X)
      y_test_pred = c.predict(X_test)
      c_o = clf(n_estimators=n, random_state=random_state)
      c_o.fit(X_o, y_o)
      y_train_pred_o = c_o.predict(X)
      y_test_pred_o = c_o.predict(X_test)
      c_name = str(clf).split(".")[-1][:-2]

      if n == n_estimators[0]:
        train_metrics[c_name] = [f1_score(y, y_train_pred, average
          ="macro")]
      else:
        train_metrics[c_name].append(f1_score(y, y_train_pred,
          average="macro"))
      if n == n_estimators[0]:
        test_metrics[c_name] = [f1_score(y_test, y_test_pred,
          average="macro")]
      else:
        test_metrics[c_name].append(f1_score(y_test, y_test_pred,
          average="macro"))

      if n == n_estimators[0]:
        train_metrics[c_name + "_o"] = [f1_score(y, y_train_pred_o
          , average="macro")]
      else:
        train_metrics[c_name + "_o"].append(f1_score(y,
          y_train_pred_o, average="macro"))
      if n == n_estimators[0]:
        test_metrics[c_name + "_o"] = [f1_score(y_test,
          y_test_pred_o, average="macro")]
      else:
        test_metrics[c_name + "_o"].append(f1_score(y_test,
          y_test_pred_o, average="macro"))

  return test_metrics, train_metrics

ctm, ctrm = comparison_oversample(X_train, y_train, X_test, y_test
```

```
    ,
                                        list(range(100, 1001, 100)),
                                        [RandomForestClassifier,
                                           AdaBoostClassifier],
                                           RANDOM_STATE)

for i in ctm:
  print(i, max(ctm[i]))

fig = plt.figure(figsize=(12, 8))
ax = plt.subplot(111)
colors = ['blue', 'green', "brown", "black"]

for i, j, c in zip(ctrm, ctm, colors):
  ax.plot(list(range(100, 1001, 100)), ctrm[i], label="{} on
     training".format(str(j)), linestyle="--", color=c)
  ax.plot(list(range(100, 1001, 100)), ctm[j], label="{} on test".
     format(str(i)), color=c)

ax.legend()
lgd = ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))
ax.set_ylabel("F1 score")
ax.set_xlabel("Number of estimators")
ax.set_title("Ensemble methods test set f1 score")
fig.savefig("ensembleoversample.png", bbox_extra_artists=(lgd,),
   bbox_inches='tight')
#files.download("ensembleoversample.png")

def compare_oversample(X, y, X_test, y_test, n_estimators, clfs,
   random_state=0):
  test_metrics = {}
  train_metrics = {}
  X_o, y_o = make_over_sample(X, y, random_state=RANDOM_STATE)
  for clf in clfs:
    for n in n_estimators:
      c = clf(n_estimators=n, random_state=random_state)
      c.fit(X, y)
      print(str(clf).split(".")[-1][:-2], n)
      y_train_pred = c.predict(X)
      y_test_pred = c.predict(X_test)
      c_o = clf(n_estimators=n, random_state=random_state)
      c_o.fit(X_o, y_o)
```

```
        y_train_pred_o = c_o.predict(X)
        y_test_pred_o = c_o.predict(X_test)
        c_name = str(clf).split(".")[-1][:-2]

        if n == n_estimators[0]:
          train_metrics[c_name] = [f1_score(y, y_train_pred, average
            ="macro")]
        else:
          train_metrics[c_name].append(f1_score(y, y_train_pred,
            average="macro"))
        if n == n_estimators[0]:
          test_metrics[c_name] = [f1_score(y_test, y_test_pred,
            average="macro")]
        else:
          test_metrics[c_name].append(f1_score(y_test, y_test_pred,
            average="macro"))

        if n == n_estimators[0]:
          train_metrics[c_name + "_o"] = [f1_score(y, y_train_pred_o
            , average="macro")]
        else:
          train_metrics[c_name + "_o"].append(f1_score(y,
            y_train_pred_o, average="macro"))
        if n == n_estimators[0]:
          test_metrics[c_name + "_o"] = [f1_score(y_test,
            y_test_pred_o, average="macro")]
        else:
          test_metrics[c_name + "_o"].append(f1_score(y_test,
            y_test_pred_o, average="macro"))

  return test_metrics, train_metrics

lgr = LogisticRegression(multi_class="multinomial", solver='newton
    -cg', random_state=RANDOM_STATE)

lgr_param = {"penalty":["l2"],
             "C":[0.001, 0.01, 0.1, 1, 10, 100]}

lgr_grd = GridSearchCV(lgr, param_grid=lgr_param, cv=5, verbose=3,
     scoring="f1_macro")

lgr_grd.fit(X_train, y_train)
```

```
print(f1_score(y_train, lgr_grd.predict(X_train), average="macro")
    , f1_score(y_test, lgr_grd.predict(X_test), average="macro"))

X_o, y_o = make_over_sample(X, y, random_state=RANDOM_STATE)

lgr_grd_o = GridSearchCV(lgr, param_grid=lgr_param, cv=5, verbose
    =3, scoring="f1_macro")

lgr_grd_o.fit(X_o, y_o)

print(f1_score(y_train, lgr_grd_o.predict(X_train), average="macro
    "), f1_score(y_test, lgr_grd_o.predict(X_test), average="macro"
    ))

svc_param = {"C":[0.001, 0.01, 0.1, 1, 10, 100],
             "degree":[1, 2, 3, 5, 6],
             "gamma":[0.0001, 0.001, 0.01, 0.1],
             "kernel":["linear", "poly", "rbf", "sigmoid"]}

svc_clf = SVC(random_state=RANDOM_STATE)
svc_grid = GridSearchCV(svc_clf, param_grid=svc_param, cv=5,
    verbose=3, scoring="f1_macro")

svc_grid.fit(X_train, y_train)

svc_grid.best_params_

print(f1_score(y_train, svc_grid.predict(X_train), average="macro"
    ), f1_score(y_test, svc_grid.predict(X_test), average="macro"))

svc_grid.fit(X_o, y_o)

svc_grid.best_params_

print(f1_score(y_train, svc_grid.predict(X_train), average="macro"
    ), f1_score(y_test, svc_grid.predict(X_test), average="macro"))

"""## Under-sampling"""

#CondensedNearestNeighbour, NeighbourhoodCleaningRule,
    RandomUnderSampler
```

```
def comparison_undersample(X, y, X_test, y_test, n_estimators,
    method, clfs, random_state=0):
  test_metrics = {}
  train_metrics = {}
  X_u, y_u = make_under_sample(X, y, method=method, random_state=
      RANDOM_STATE)
  for i, clf in enumerate(clfs):
    for n in n_estimators:
      c = clf(n_estimators=n, random_state=random_state)
      c.fit(X, y)
      print(str(clf).split(".")[-1][:-2], n)
      y_train_pred = c.predict(X)
      y_test_pred = c.predict(X_test)
      c_u = clf(n_estimators=n, random_state=random_state)
      c_u.fit(X_u, y_u)
      y_train_pred_u = c_u.predict(X)
      y_test_pred_u = c_u.predict(X_test)
      c_name = str(clf).split(".")[-1][:-2]

      if n == n_estimators[0]:
        train_metrics[c_name] = [f1_score(y, y_train_pred, average
            ="macro")]
      else:
        train_metrics[c_name].append(f1_score(y, y_train_pred,
            average="macro"))
      if n == n_estimators[0]:
        test_metrics[c_name] = [f1_score(y_test, y_test_pred,
            average="macro")]
      else:
        test_metrics[c_name].append(f1_score(y_test, y_test_pred,
            average="macro"))

      if n == n_estimators[0]:
        train_metrics[c_name + "_u"] = [f1_score(y, y_train_pred_u
            , average="macro")]
      else:
        train_metrics[c_name + "_u"].append(f1_score(y,
            y_train_pred_u, average="macro"))
      if n == n_estimators[0]:
        test_metrics[c_name + "_u"] = [f1_score(y_test,
            y_test_pred_u, average="macro")]
      else:
```

```
        test_metrics[c_name + "_u"].append(f1_score(y_test,
            y_test_pred_u, average="macro"))

    return test_metrics, train_metrics

runtm, runtrm = comparison_undersample(X_train, y_train, X_test,
    y_test,
                                       list(range(100, 1001, 100))
                                           ,
                                       RandomUnderSampler,
                                       [RandomForestClassifier,
                                           AdaBoostClassifier],
                                           RANDOM_STATE)

fig = plt.figure(figsize=(12, 8))
ax = plt.subplot(111)
colors = ['blue', 'green', "brown", "black"]

for i, j, c in zip(runtrm, runtm, colors):
    ax.plot(list(range(100, 1001, 100)), runtrm[i], label="{} on
        training".format(str(j)), linestyle="--", color=c)
    ax.plot(list(range(100, 1001, 100)), runtm[j], label="{} on test
        ".format(str(i)), color=c)

ax.legend()
lgd = ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))
ax.set_ylabel("F1 score")
ax.set_xlabel("Number of estimators")
ax.set_title("Ensemble methods test set f1 score")
fig.savefig("ensembleundersample.png", bbox_extra_artists=(lgd,),
    bbox_inches='tight')
#files.download("ensembleundersample.png")

X_u, y_u = make_under_sample(X_train, y_train, method=
    NeighbourhoodCleaningRule, random_state=RANDOM_STATE)
print(len(Counter(y_u)))
Counter(y_u)

cnntm, cnntrm = comparison_undersample(X_train, y_train, X_test,
    y_test,
                                       list(range(100, 1001, 100))
                                           ,
```

```
                                                CondensedNearestNeighbour,
                                                [RandomForestClassifier,
                                                    AdaBoostClassifier],
                                                RANDOM_STATE)

fig = plt.figure(figsize=(12, 8))
ax = plt.subplot(111)
colors = ['blue', 'green', "brown", "black"]

for i, j, c in zip(cnntrm, cnntm, colors):
  ax.plot(list(range(100, 1001, 100)), cnntrm[i], label="{} on
      training".format(str(j)), linestyle="--", color=c)
  ax.plot(list(range(100, 1001, 100)), cnntm[j], label="{} on test
      ".format(str(i)), color=c)

ax.legend()
lgd = ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))
ax.set_ylabel("F1 score")
ax.set_xlabel("Number of estimators")
ax.set_title("Ensemble methods test set f1 score")
fig.savefig("ensembleundersample.png", bbox_extra_artists=(lgd,),
    bbox_inches='tight')
#files.download("ensembleundersample.png")

ncltm, ncltrm = comparison_undersample(X_train, y_train, X_test,
    y_test,
                                                list(range(100, 1001, 100))
                                                    ,
                                                NeighbourhoodCleaningRule,
                                                [RandomForestClassifier,
                                                    AdaBoostClassifier],
                                                RANDOM_STATE)

fig = plt.figure(figsize=(12, 8))
ax = plt.subplot(111)
colors = ['blue', 'green', "brown", "black"]

for i, j, c in zip(ncltrm, ncltm, colors):
  ax.plot(list(range(100, 1001, 100)), ncltrm[i], label="{} on
      training".format(str(j)), linestyle="--", color=c)
  ax.plot(list(range(100, 1001, 100)), ncltm[j], label="{} on test
      ".format(str(i)), color=c)
```

```
ax.legend()
lgd = ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))
ax.set_ylabel("F1 score")
ax.set_xlabel("Number of estimators")
ax.set_title("Ensemble methods test set f1 score")
fig.savefig("ensembleundersample.png", bbox_extra_artists=(lgd,),
    bbox_inches='tight')
#files.download("ensembleundersample.png")

svc_param = {"C":[0.001, 0.01, 0.1, 1, 10, 100],
             "degree":[1, 2, 3, 5, 6],
             "gamma":[0.0001, 0.001, 0.01, 0.1],
             "kernel":["linear", "poly", "rbf", "sigmoid"]}

svc_grid_base = GridSearchCV(svc_clf, param_grid=svc_param, cv=5,
    verbose=3, scoring="f1_macro")

svc_grid_base.fit(X_train, y_train)

print(f1_score(y_train, svc_grid_base.predict(X_train), average="
    macro"), f1_score(y_test, svc_grid_base.predict(X_test),
    average="macro"))

svc_grid_rus = GridSearchCV(svc_clf, param_grid=svc_param, cv=3,
    verbose=3, scoring="f1_macro")

X_u, y_u = make_under_sample(X_train, y_train, method=
    RandomUnderSampler, random_state=RANDOM_STATE)

svc_grid_rus.fit(X_u, y_u)

print(f1_score(y_train, svc_grid_rus.predict(X_train), average="
    macro"), f1_score(y_test, svc_grid_rus.predict(X_test), average
    ="macro"))

svc_grid_cnn = GridSearchCV(svc_clf, param_grid=svc_param, cv=3,
    verbose=1, scoring="f1_macro")

X_u, y_u = make_under_sample(X_train, y_train, method=
    CondensedNearestNeighbour, random_state=RANDOM_STATE)
```

```
svc_grid_cnn.fit(X_u, y_u)

print(f1_score(y_train, svc_grid_cnn.predict(X_train), average="
    macro"), f1_score(y_test, svc_grid_cnn.predict(X_test), average
    ="macro"))

svc_grid_ncl = GridSearchCV(svc_clf, param_grid=svc_param, cv=3,
    verbose=1, scoring="f1_macro")

X_u, y_u = make_under_sample(X_train, y_train, method=
    NeighbourhoodCleaningRule, random_state=RANDOM_STATE)

svc_grid_ncl.fit(X_u, y_u)

print(f1_score(y_train, svc_grid_ncl.predict(X_train), average="
    macro"), f1_score(y_test, svc_grid_ncl.predict(X_test), average
    ="macro"))

lgr_grd_rus = GridSearchCV(lgr, param_grid=lgr_param, cv=3,
    verbose=1, scoring="f1_macro")

X_u, y_u = make_under_sample(X_train, y_train, method=
    RandomUnderSampler, random_state=RANDOM_STATE)

lgr_grd_rus.fit(X_u, y_u)

print(f1_score(y_train, lgr_grd_rus.predict(X_train), average="
    macro"), f1_score(y_test, lgr_grd_rus.predict(X_test), average=
    "macro"))

lgr_grd_cnn = GridSearchCV(lgr, param_grid=lgr_param, cv=3,
    verbose=1, scoring="f1_macro")

X_u, y_u = make_under_sample(X_train, y_train, method=
    CondensedNearestNeighbour, random_state=RANDOM_STATE)

lgr_grd_cnn.fit(X_u, y_u)

print(f1_score(y_train, lgr_grd_cnn.predict(X_train), average="
    macro"), f1_score(y_test, lgr_grd_cnn.predict(X_test), average=
    "macro"))
```

```
lgr_grd_ncl = GridSearchCV(lgr, param_grid=lgr_param, cv=3,
    verbose=1, scoring="f1_macro")

X_u, y_u = make_under_sample(X_train, y_train, method=
    NeighbourhoodCleaningRule, random_state=RANDOM_STATE)

lgr_grd_ncl.fit(X_u, y_u)

print(f1_score(y_train, lgr_grd_ncl.predict(X_train), average="
    macro"), f1_score(y_test, lgr_grd_ncl.predict(X_test), average=
    "macro"))

"""## Scaling"""

def scaled_data(X, y):
  scalar = MinMaxScaler()
  X = scalar.fit_transform(X)
  return X, y

def comparison_scaled(X, y, X_test, y_test, n_estimators, clfs,
    random_state=0):
  test_metrics = {}
  train_metrics = {}
  X_s, y_s = scaled_data(X, y)
  for i, clf in enumerate(clfs):
    for n in n_estimators:
      c = clf(n_estimators=n, random_state=random_state)
      c.fit(X, y)
      print(str(clf).split(".")[-1][:-2], n)
      y_train_pred = c.predict(X)
      y_test_pred = c.predict(X_test)
      c_s = clf(n_estimators=n, random_state=random_state)
      c_s.fit(X_s, y_s)
      y_train_pred_s = c_s.predict(X)
      y_test_pred_s = c_s.predict(X_test)
      c_name = str(clf).split(".")[-1][:-2]

      if n == n_estimators[0]:
        train_metrics[c_name] = [f1_score(y, y_train_pred, average
          ="macro")]
      else:
        train_metrics[c_name].append(f1_score(y, y_train_pred,
```

```
            average="macro"))
        if n == n_estimators[0]:
            test_metrics[c_name] = [f1_score(y_test, y_test_pred,
                average="macro")]
        else:
            test_metrics[c_name].append(f1_score(y_test, y_test_pred,
                average="macro"))

        if n == n_estimators[0]:
            train_metrics[c_name + "_s"] = [f1_score(y, y_train_pred_s
                , average="macro")]
        else:
            train_metrics[c_name + "_s"].append(f1_score(y,
                y_train_pred_s, average="macro"))
        if n == n_estimators[0]:
            test_metrics[c_name + "_s"] = [f1_score(y_test,
                y_test_pred_s, average="macro")]
        else:
            test_metrics[c_name + "_s"].append(f1_score(y_test,
                y_test_pred_s, average="macro"))

    return test_metrics, train_metrics

stm, strm = comparison_scaled(X_train, y_train, X_test, y_test,
                              list(range(100, 1001, 100)),
                              [RandomForestClassifier,
                                  AdaBoostClassifier], RANDOM_STATE
                                  )

fig = plt.figure(figsize=(12, 8))
ax = plt.subplot(111)
colors = ['blue', 'green', "brown", "black"]

for i, j, c in zip(strm, stm, colors):
    ax.plot(list(range(100, 1001, 100)), strm[i], label="{} on
        training".format(str(j)), linestyle="--", color=c)
    ax.plot(list(range(100, 1001, 100)), stm[j], label="{} on test".
        format(str(i)), color=c)

ax.legend()
lgd = ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))
ax.set_ylabel("F1 score")
```

```
ax.set_xlabel("Number of estimators")
ax.set_title("Ensemble methods test set f1 score")
fig.savefig("scalingcompare.png", bbox_extra_artists=(lgd,),
    bbox_inches='tight')
#files.download("scalingcompare.png")



X_s, y_s = scaled_data(X_train, y_train)

rf_S = RandomForestClassifier(n_estimators=800)

rf_S.fit(X_s, y_s)

y_pred_s = rf_S.predict(X_test)

c = confusion_matrix(y_test, y_pred_s)

cn = confusion_matrix(y_test, RandomForestClassifier(n_estimators
    =800).fit(X_train, y_train).predict(X_test))

hm = sns.heatmap(c.T, square=True, annot=True, fmt="d", cbar=False
    , cmap='coolwarm')
plt.xlabel("true labels")
plt.ylabel("predicted labels")
#hm.get_figure().savefig("rfheatmap.png")
#files.download("rfheatmap.png")

fig = plt.figure(figsize=(12, 8))

ax = plt.subplot(121)
hm = sns.heatmap(c.T, square=True, annot=True, fmt="d", cbar=False
    , cmap='coolwarm')
plt.xlabel("true labels")
plt.ylabel("predicted labels")
plt.title("Confusion matrix for scaled data")
ax = plt.subplot(122)
hmn = sns.heatmap(cn.T, square=True, annot=True, fmt="d", cbar=
    False, cmap='coolwarm')
plt.xlabel("true labels")
plt.ylabel("predicted labels")
plt.title("Confusion matrix for baseline data")
```