UNIVERSITÀ DEGLI STUDI DI BRESCIA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea Magistrale in Ingegneria Informatica

Study of Various Machine Learning Algorithms Applied to the Online Shoppers Purchasing Intention Data Set

Teachers:

Ch.mo Prof. Alfonso E. Gerevini

Ch.mo Prof. Ivan Serina

Students:

Matteo Beatrice (739848)

Luca Cotti (719204)

Academic Year 2021/2022

Contents

Introduction

The objective of this project is to compare various machine learning classification algorithms (in both their basic form and also when combined in ensembles), when applied to the *Online Shoppers Purchasing Intention Data Set* [1].

The data set, which was originally conceived for a study on real-time prediction of online shoppers' purchasing intention [2], contains information about the visitors of a shopping website, including metrics taken from Google Analytics. The objective is to predict whether a certain visitor will generate revenue or not: this could allow to offer certain content only to those who intend to purchase and not to the other users.

The data set was formed so that each session would belong to a different user in a 1-year period to avoid any tendency to a specific campaign, special day, user profile, or period.

The original authors specified that the data set is imbalanced, so special care will be required to avoid excessive bias towards the majority class.

The project begins with an initial setup in chapter 1, where we import and analyze the data set. Then we proceed with a cleaning and with feature engineering phase.
In chapter 2, we will develop and optimize various basic classification models, built with *sci-kit learn* [3].
In chapter 3, we will try to to build ensembles of basic classifiers, to create more accurate models. We will once again use *sci-kit learn*, along with the ensembles provided by *imbalanced-learn* [4].
Chapter 4 describes the development and optimization of a (rather basic) neural network, built with *Keras* [5].
Finally, in chapter 5, we report the results obtained by *Auto-sklearn* [6] (in particular, we will use *Auto-sklearn 2* [7]), and compare them with our conclusions.

1. Initial Setup

1.1 Imports

We need to import a few common modules, initialize random seeds, ensure Matplotlib plots figures are inline and we also need to prepare a function to save the figures. We also check that Python 3.5 or later is installed, as well as Scikit-Learn $\geq$ 0.20.

```
import matplotlib.pyplot as plt
import matplotlib as mpl
import pandas as pd
import os
import tensorflow as tf
import numpy as np
import sklearn
import sys

# Python >= 3.5 is required
assert sys.version_info >= (3, 5)
# Scikit-Learn >= 0.20 is required
assert sklearn.__version__ >= "0.20"

# Initialize random seeds
np.random.seed(42)
tf.random.set_seed(42)

# To plot pretty figures
%matplotlib inline
mpl.rc("axes", labelsize=14)
mpl.rc("xtick", labelsize=12)
mpl.rc("ytick", labelsize=12)

# Where to save the figures
```

```
ROOT_DIR = "."
IMAGES_PATH = os.path.join(ROOT_DIR, "images")
os.makedirs(IMAGES_PATH, exist_ok=True)


def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
    """ Saves a figure """
    path = os.path.join(IMAGES_PATH, fig_id + "." + fig_extension)
    print("Saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

1.2 Download Dataset

Since the data set is provided as a `.csv` file, we can use the `read_csv` function of `pandas` to import it directly from the URL.

```
DATASET_URL = "https://archive.ics.uci.edu/ml/machine-learning-databases/
 ↪00468/online_shoppers_intention.csv"
dataset = pd.read_csv(DATASET_URL)
```

We can use the `info` method to print a concise summary of the imported data set.

```
dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 12330 entries, 0 to 12329
Data columns (total 18 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   Administrative           12330 non-null  int64
 1   Administrative_Duration  12330 non-null  float64
 2   Informational            12330 non-null  int64
 3   Informational_Duration   12330 non-null  float64
 4   ProductRelated           12330 non-null  int64
 5   ProductRelated_Duration  12330 non-null  float64
 6   BounceRates              12330 non-null  float64
 7   ExitRates                12330 non-null  float64
 8   PageValues               12330 non-null  float64
 9   SpecialDay               12330 non-null  float64
 10  Month                    12330 non-null  object
 11  OperatingSystems         12330 non-null  int64
 12  Browser                  12330 non-null  int64
 13  Region                   12330 non-null  int64
 14  TrafficType              12330 non-null  int64
 15  VisitorType              12330 non-null  object
 16  Weekend                  12330 non-null  bool
 17  Revenue                  12330 non-null  bool
dtypes: bool(2), float64(7), int64(7), object(2)
memory usage: 1.5+ MB
```

The data set contains a total of `12330` entries, with `18` features. Some features are numeric, others are categorical.

`Administrative`, `Administrative Duration`, `Informational`, `Informational Duration`, `Product Related` and `Product Related Duration` represent the number of different types of pages visited by the visitor in that session and total time spent in each of these page categories.

The values of these features are derived from the URL information of the pages visited by the user and updated in real time when a user takes an action, e.g. moving from one page to another.

The `Bounce Rate`, `Exit Rate` and `Page Value` features represent the metrics measured by Google Analytics for each page in the e-commerce site.

The value of `Bounce Rate` feature for a web page refers to the percentage of visitors who enter the site from that page and then leave ("bounce") without triggering any other requests to the analytics server during that session.

The value of `Exit Rate` feature for a specific web page is calculated as for all pageviews to the page, the percentage that were the last in the session.

The `Page Value` feature represents the average value for a web page that a user visited before completing an e-commerce transaction.

The `Special Day` feature indicates the closeness of the site visiting time to a specific special day (e.g. Mother's Day, Valentine's Day) in which the sessions are more likely to be finalized with transaction. The value of this attribute is determined by considering the dynamics of e-commerce such as the duration between the order date and delivery date. For example, for Valentine's day, this value takes a nonzero value between February 2 and February 12, zero before and after this date unless it is close to another special day, and its maximum value of 1 on February 8.

`Revenue` represents the class of the instance: a `True` value means the user generated revenue, and a `False` value means the user did not generate revenue.

The data set also includes operating system, browser, region, traffic type, visitor type as returning or new visitor, a Boolean value indicating whether the date of the visit is weekend, and month of the year.

We can get a glimpse of the data by using the `head` method.

```
dataset.head()
```

```
   Administrative  Administrative_Duration  BounceRates   ...  Revenue
0               0                      0.0         0.20   ...    False
1               0                      0.0         0.00   ...    False
2               0                      0.0         0.20   ...    False
3               0                      0.0         0.05   ...    False
4               0                      0.0         0.02   ...    False
```

It's also important to check if the number of instances in each class is balanced:

```
dataset["Revenue"].value_counts(normalize=True)
```

```
False    0.845255
True     0.154745
Name: Revenue, dtype: float64
```

The `84.5%` of instances are negative, while the the `15.5%` are positive. This means that the data set is imbalanced.

There are a few different ways to handle imbalanced data sets, such as *under-sampling* the majority class, or *oversampling* the minority one. We could also use more advanced algorithms, such as *SMOTE*, to generate synthetic samples from the minority class.

In this project, we will not make use of such techniques directly, but we will tune the algorithms to account for the imbalance as much as possible, and we will also use algorithms created specifically for imbalanced data sets. Most of these algorithms do use one (or more) of the methods mentioned above.

Rather than the basic *accuracy* metric, we will use *balanced accuracy*, which is suited for imbalanced data. *Balanced accuracy* is defined as the arithmetic mean of *accuracy* and *recall*:

$$\text{Balanced Accuracy} = \frac{sensitivity + specificity}{2}$$

We could also use the *F1 score* metric, which is the harmonic mean of *precision* and *recall*:

$$\text{F1 Score} = 2 * \frac{precision * sensitivity}{precision + sensitivity}$$

Where *sensitivity* is the proportion of actual positives that are correctly identified as such, *specificity* is the proportion of actual negatives that are correctly identified, and *precision* quantifies the number of correct positive predictions made out of positive predictions made by the model.

$$\text{Sensitivity} = \frac{TP}{TP + FN} \qquad \text{Specificity} = \frac{TN}{TN + FP} \qquad \text{Precision} = \frac{TP}{TP + FP}$$

The *F1 score* however doesn't care about how many true negatives are being classified. For the purposes of this project positives are as import as negatives, so *balanced accuracy* is a better metric.

1.3 Data Cleaning and Feature Engineering

1.3.1 Column Names

The column naming convention appears to be inconsistent. We can begin the data cleaning process by converting all the column names to *snake_ case*.

```python
def to_snake_case(str):
    res = [str[0].lower()]
    for i, c in enumerate(str[1:]):
        if c in ('ABCDEFGHIJKLMNOPQRSTUVWXYZ'):
            if str[i] != "_":
                res.append('_')
            res.append(c.lower())
        else:
            res.append(c)

    return ''.join(res)
```

```python
dataset.columns = dataset.columns.map(lambda c: to_snake_case(c))
```

1.3.2 Column Types

There are a few columns that represent categorical data, and two boolean columns that could cause problems. We can convert such columns to more convenient data types.

Convert categorical data

The data set has two categorical features with string values: `Month` and `VisitorType`. We can observe the possible values of these features.

```python
print(dataset["month"].unique())
print(dataset["visitor_type"].unique())
```

```
['Feb' 'Mar' 'May' 'Oct' 'June' 'Jul' 'Aug' 'Nov' 'Sep' 'Dec']
['Returning_Visitor' 'New_Visitor' 'Other']
```

These features can be converted into integers using `sklearn.preprocessing.LabelEncoder()`, which replaces the category string values with increasing integers values.

The problem with this method is that the learning algorithms could interpret the integer values as having an order/hierarchy between them. This is fine for the `Month` column, where the various months will be encoded with integers between 0 and 11: these integers do have a meaningful order, so label encoding is correct.

However for `VisitorType` the integer values that the feature could have to represent the string values have no order/hierarchy. For this reason, a more appropriate conversion for this feature would be to use one-hot encoding. In this strategy, each possible category value is converted into a new column and assigned a 1 or 0 value depending on the value in the original column.

```python
from sklearn.preprocessing import LabelEncoder


le = LabelEncoder()
dataset["month"] = le.fit_transform(dataset["month"])


dum_df = pd.get_dummies(dataset["visitor_type"], prefix="visitor_type")
dum_df.columns = dum_df.columns.map(lambda c: to_snake_case(c))


dataset = dataset.join(dum_df).drop("visitor_type", axis=1)
```

Convert Boolean Values to Integer

The `weekend` and `revenue` columns have boolean values, which should automatically be converted to `0` and `1` by Python, but since some of the algorithms may have some parts implemented in C/C++, we might run into some problems.

To avoid this, we can convert such columns to integer values.

```python
dataset["weekend"] = dataset["weekend"].astype(int)
dataset["revenue"] = dataset["revenue"].astype(int)
```

1.3.3 Missing Values

We can check if the dataset contains NA values, and if it does we can delete such rows as they might ruin the learning process.

```
dataset.isna().any()
```

```
administrative                    False
administrative_duration           False
informational                     False
informational_duration            False
product_related                   False
product_related_duration          False
bounce_rates                      False
exit_rates                        False
page_values                       False
special_day                       False
month                             False
operating_systems                 False
browser                           False
region                            False
traffic_type                      False
weekend                           False
revenue                           False
visitor_type_new_visitor          False
visitor_type_other                False
visitor_type_returning_visitor    False
dtype: bool
```

The dataset does not contain any NA value, so no rows need to be eliminated.

1.3.4 Correlated Columns

To increase the speed of the learning process and reduce bias a possibility is to remove highly correlated columns from the dataset.

However, while developing this project, we discovered *Recursive Feature Elimination*, which provides a more effective way of removing unnecessary features.

We are leaving this section here anyway for compleness.

```
def get_correlated_cols(ds: pd.DataFrame, corr_threshold: float):
    # Compute correlation matrix using pearson method (linear correlation)
    corr = ds.corr(method="pearson")
    # Find collinear columns
    corr_cols = corr[corr > corr_threshold].dropna(
        thresh=2).dropna(axis="columns")
    return corr_cols
```

```
get_correlated_cols(dataset, 0.90)
```

```
              bounce_rates  exit_rates
bounce_rates      1.000000    0.913004
exit_rates        0.913004    1.000000
```

`bounce_rates` and `exit_rates` are highly correlated. We can remove one of the two columns.

```
# dataset.drop("exit_rates", axis=1, inplace=True)
```

1.3.5 Duplicate Rows

It's good practice to identify and remove duplicate rows in the data set, because they could result in misleading performance when evaluating ML algorithms: duplicate rows could appear in both train and test data sets.

```
dups = dataset.duplicated()
print(dups.any())
```

```
True
```

There are duplicates in the data set. They can be removed using pandas `drop_duplicates`.

```
dataset.drop_duplicates(inplace=True)
```

1.3.6 Feature Scaling

In order to weight the features equally, feature scaling is important. Here we use the `MinMaxScaler`, which scales all values to the [0,1] interval.

```
from pandas.core.common import random_state
from sklearn.preprocessing import MinMaxScaler

mms = MinMaxScaler()
dataset = pd.DataFrame(mms.fit_transform(dataset), columns=dataset.columns)
```

1.3.7 Create Testing and Training sets

We will use the 70% of the total instances for training, and the remaining 30% for testing.

The `stratify` option of `train_test_split` ensures that relative class frequencies are approximately preserved in the training and test sets.

```
from sklearn.model_selection import train_test_split

X = dataset.drop("revenue", axis=1)
y = dataset["revenue"].copy()

X_train, X_test, y_train,  y_test = train_test_split(
    X, y, test_size=0.3, stratify=y, random_state=42)
```

1.3.8 Recursive Feature Elimination

Recursive Feature Elimination (RFE) is a feature selection algorithms. Feature selection refers to techniques that select a subset of the most relevant features (columns) for a data set. Fewer features can allow machine learning algorithms to run more efficiently (less space or time complexity) and be more effective. Some machine learning algorithms can be misled by irrelevant input features, resulting in worse predictive performance.

RFE in particular searches for a subset of features by starting with all features in the training data set and successfully removing features until the desired number remains.

This is achieved by fitting a given machine learning algorithm, ranking features by importance, discarding the least important features, and re-fitting the model. This process is repeated until a specified number of features remains.

Choosing the optimal number of features to keep is not trivial: `scikit-learn` provides the

RFECV class, which performs cross-validation evaluation of different numbers of features and automatically selects the features that resulted in the best mean score.

Since we know that the data set is imbalanced, we use a random forest classifier to compute the weights associated with the features (the features with the lowest weights are those that will be removed), along with the balanced_accuracy to measure performance during the cross-validation.

```python
from sklearn.feature_selection import RFECV
from sklearn.ensemble import RandomForestClassifier

rfecv = RFECV(
    estimator=RandomForestClassifier(random_state=42),
    min_features_to_select=1,  # Eliminate at least one feature
    scoring="balanced_accuracy"
)
rfecv.fit(X_train, y_train)

print("Eliminated %d features, from %d features to %d features" % (
    len(X.columns) - rfecv.n_features_,
    len(X.columns),
    rfecv.n_features_
)
)

columns_to_keep = X.columns[rfecv.support_]
X = X[columns_to_keep]
X_train = X_train[columns_to_keep]
X_test = X_test[columns_to_keep]
```

```
Eliminated 12 features, from 19 features to 7 features
```

1.3.9 Utility Functions

```
from sklearn.metrics import balanced_accuracy_score, confusion_matrix,␣
↪ConfusionMatrixDisplay

def print_confusion_matrix(y_true, y_pred):
    ConfusionMatrixDisplay(confusion_matrix=confusion_matrix(
        y_true, y_pred), display_labels=["No Revenue", "Revenue"]).plot()

def evaluate(clf):
    y_pred = clf.predict(X_test)
    print("Accuracy (on test set): ", balanced_accuracy_score(y_test,␣
↪y_pred))

def evaluate_grid(grid_clf):
    y_pred = grid_clf.predict(X_test)
    print("Best parameters: ", grid_clf.best_params_)
    print("Accuracy of best (means of cross-validated scores on train set):␣
↪",
          grid_clf.best_score_)
    print("Accuracy of best (on test set): ",
          balanced_accuracy_score(y_test, y_pred))
    print_confusion_matrix(y_test, y_pred)
```

2. Basic Classifiers

2.1 Decision Tree

We expect decision trees to behave well for this problem, because they are particularly suited for imbalanced classifications.

We will begin with a baseline decision tree, that we will try to improve upon by tuning the hyper-parameters.

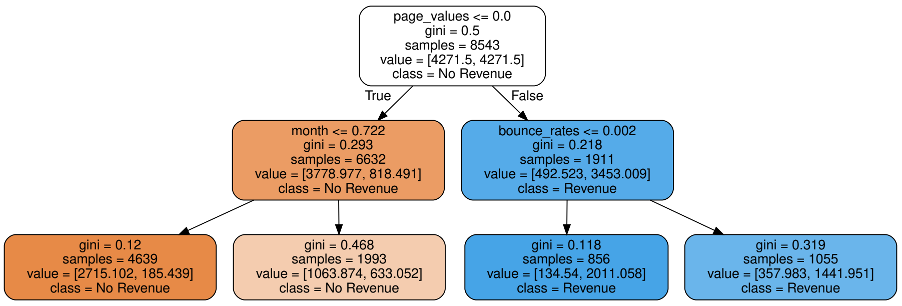
```
from sklearn.tree import DecisionTreeClassifier

# balanced weight => weights inversely proportional to class frequencies
tree_clf = DecisionTreeClassifier(
    max_depth=2, random_state=42, class_weight="balanced")
tree_clf.fit(X_train, y_train)
```

```
evaluate(tree_clf)
```

```
Accuracy (on test set):  0.8419354108674497
```

We can also visualize the decision tree:

```
from graphviz import Source
from sklearn.tree import export_graphviz

export_graphviz(
    tree_clf,
    out_file=os.path.join(IMAGES_PATH, "tree.dot"),
    feature_names=X.columns,
    class_names=["No Revenue", "Revenue"],
    filled=True,
    rounded=True,
)
Source.from_file(os.path.join(IMAGES_PATH, "tree.dot"))
```

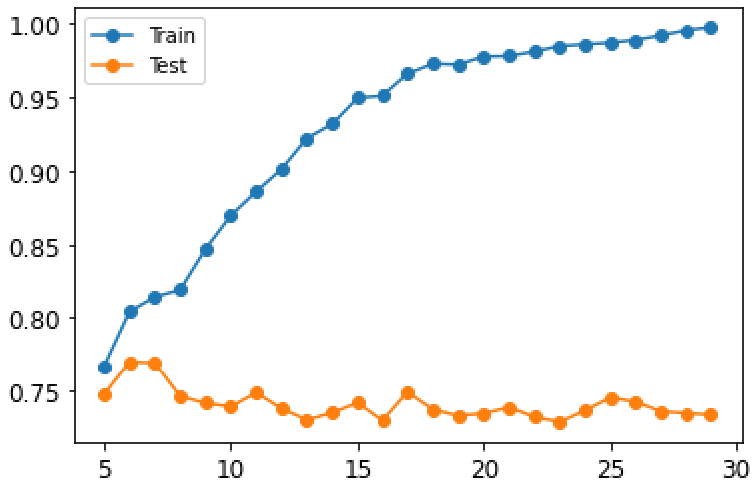


To check for over-fitting, we can plot the accuracy on the training and test sets. Here we also evaluate how the decision trees behaves when using different criteria.

```python
def evaluate_decision_tree(values, criterion="gini"):
    train_scores, test_scores = list(), list()
    for i in values:
        # configure the model
        model = DecisionTreeClassifier(
            max_depth=i, criterion=criterion, random_state=42)
        # fit model on the training dataset
        model.fit(X_train, y_train)
        # evaluate on the train dataset
        train_yhat = model.predict(X_train)
        train_acc = balanced_accuracy_score(y_train, train_yhat)
        train_scores.append(train_acc)
        # evaluate on the test dataset
        test_yhat = model.predict(X_test)
        test_acc = balanced_accuracy_score(y_test, test_yhat)
        test_scores.append(test_acc)
        # summarize progress
        print('>%d, train: %.3f, test: %.3f' % (i, train_acc, test_acc))
    # plot of train and test scores vs tree depth
    plt.plot(values, train_scores, '-o', label='Train')
    plt.plot(values, test_scores, '-o', label='Test')
    plt.legend()
    plt.show()
```

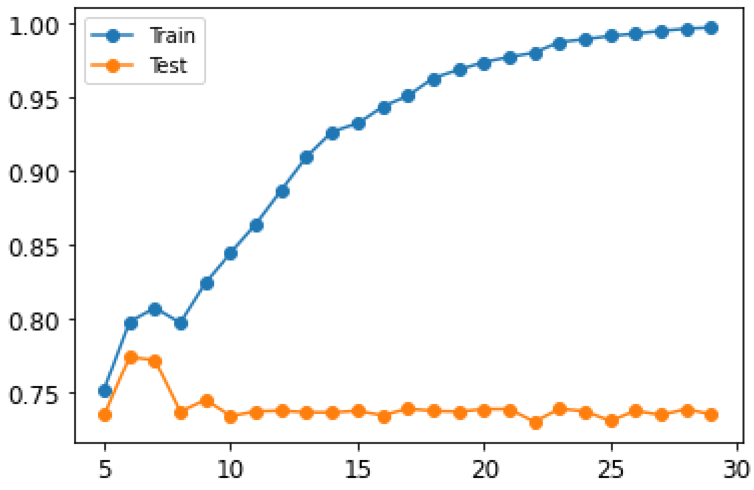
```
evaluate_decision_tree([i for i in range(5, 30)])
```

```
>5, train: 0.767, test: 0.748
...
>29, train: 0.998, test: 0.734
```



```
evaluate_decision_tree([i for i in range(5, 30)], criterion="entropy")
```

```
>5, train: 0.752, test: 0.735
...
>29, train: 0.997, test: 0.735
```



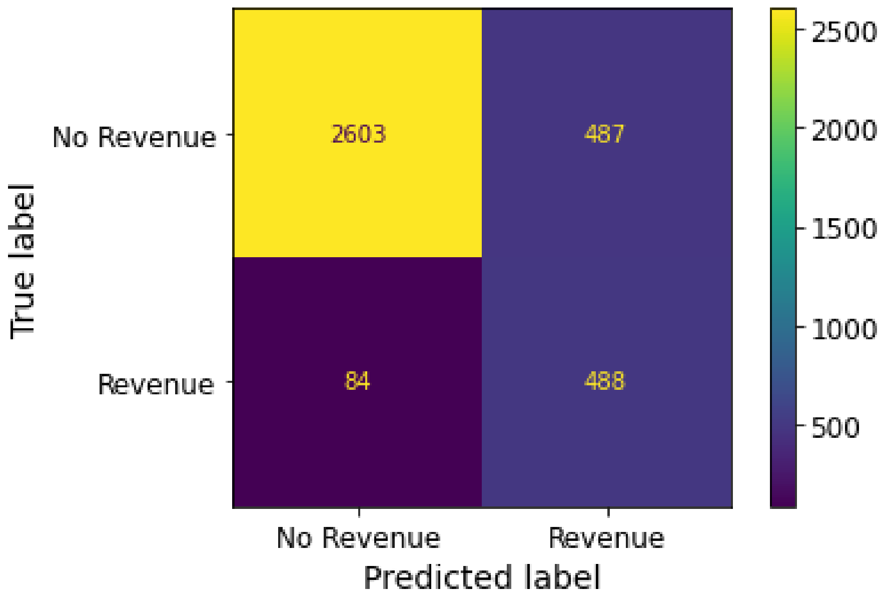
For a more comprehensive hyper-parameter tuning, `scikit-learn` provides the `GridSearchCV` class, which creates a grid search to find the best possible hyper-parameters for a model (by exhaustively trying all possible combinations of the given parameters), and evaluates each possible model through cross-validation, in order to keep over-fitting in check. The model with the highest cross-validated score will be kept as the best one.

We also evaluate the (balanced) accuracy of the best model using the test set.

```
from sklearn.model_selection import GridSearchCV


params = {
    "criterion": ["gini", "entropy"],
    "max_depth": list(range(1, 10)),
    "max_features": [None, "log2", "sqrt"],
}


grid_tree_clf = GridSearchCV(tree_clf, params, scoring="balanced_accuracy")
grid_tree_clf.fit(X_train, y_train)
```

```
evaluate_grid(grid_tree_clf)
```

```
Best parameters:  {'criterion': 'entropy', 'max_depth': 4, 'max_features':␣
 ↪None}
Accuracy of best (means of cross-validated scores on train set):
0.8501342341213902
Accuracy of best (on test set):  0.8477708375766628
```



2.2 K-Nearest Neighbor

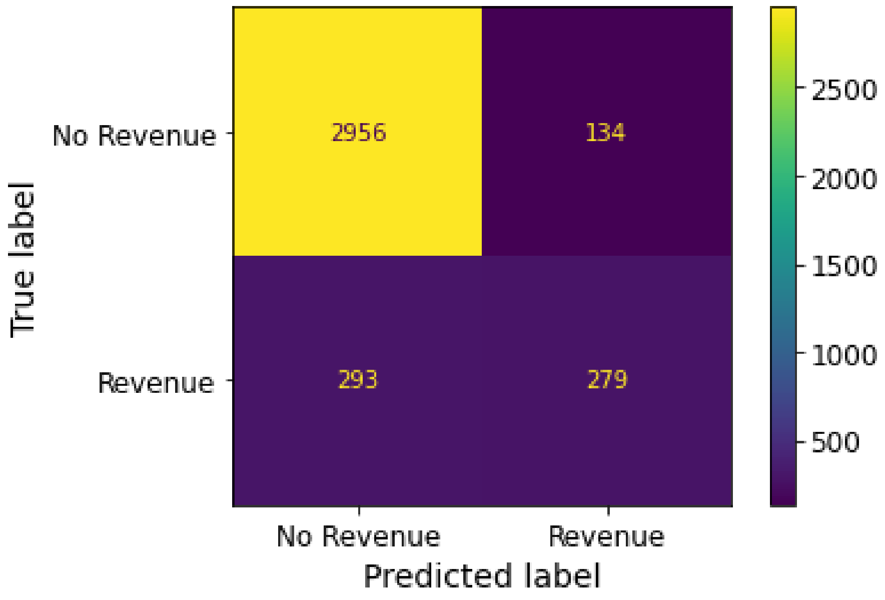
The *K-Nearest Neighbor* (KNN) algorithm (at least in its basic form) struggles with imbalanced data, but at the same it should also perform particularly well for datasets with a lower number of features. Thanks to RFE, we managed to reduce the number of features to 9, so it is interesting to verify the results of KNN.

We can also verify how the accuracy changes with various distance metrics.

```python
from sklearn.neighbors import KNeighborsClassifier

params = {
    "n_neighbors": list(range(1, 10)),
    "weights": ["uniform", "distance"],
    "metric": ["euclidean", "chebyshev", "minkowski", "manhattan"]
}

knn_clf = KNeighborsClassifier()

grid_knn_clf = GridSearchCV(knn_clf, params, scoring="balanced_accuracy")
grid_knn_clf.fit(X_train, y_train)
```

```
evaluate_grid(grid_knn_clf)
```

```
Best parameters:  {'metric': 'euclidean', 'n_neighbors': 6, 'weights':
'distance'}
Accuracy of best (means of cross-validated scores on train set):
0.7290814711036219
Accuracy of best (on test set):  0.7221982709846787
```



2.3 Logistic Regression

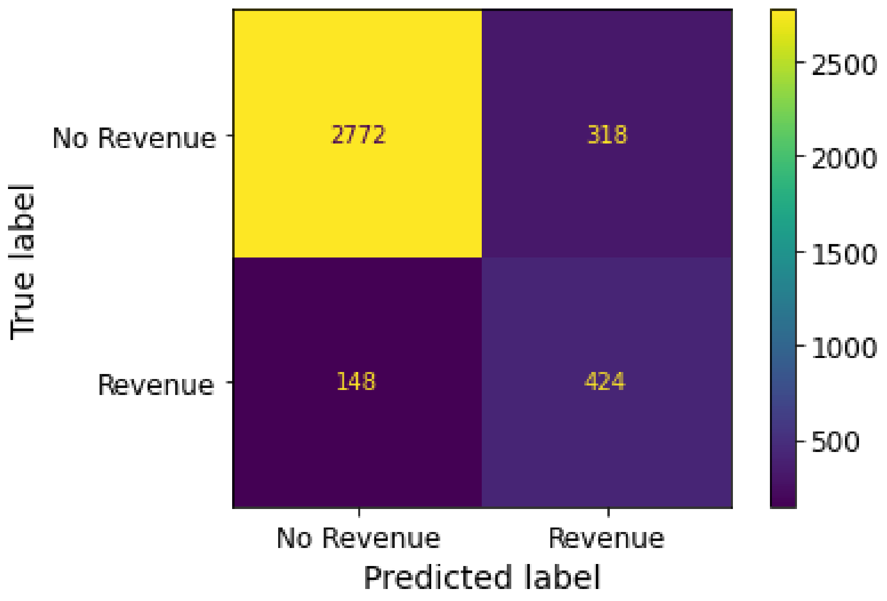
Logistic regression is especially suited for binary problems and it can also be tuned for imbalanced data (by setting `class_weight="balanced"`), so we expect this method to give good results.

The `LogisticRegressionCV` class also allows to specify lists of parameters to try. Just like the `GridSearchCV` class, it evaluates each possible combination of parameters and keeps the best one, while also having noticable performance improvements.

```python
from sklearn.linear_model import LogisticRegressionCV


log_clf = LogisticRegressionCV(
    random_state=42,
    Cs=np.logspace(-4, 4, num=30),  # 30 items in logspace from 10^-4 to␣
 ↪10^4
    scoring="balanced_accuracy",
    max_iter=500,
    class_weight="balanced"
)

log_clf.fit(X_train, y_train)
```

```
evaluate(log_clf)
print_confusion_matrix(y_test, log_clf.predict(X_test))
```

```
Accuracy (on test set):  0.819173059949759
```



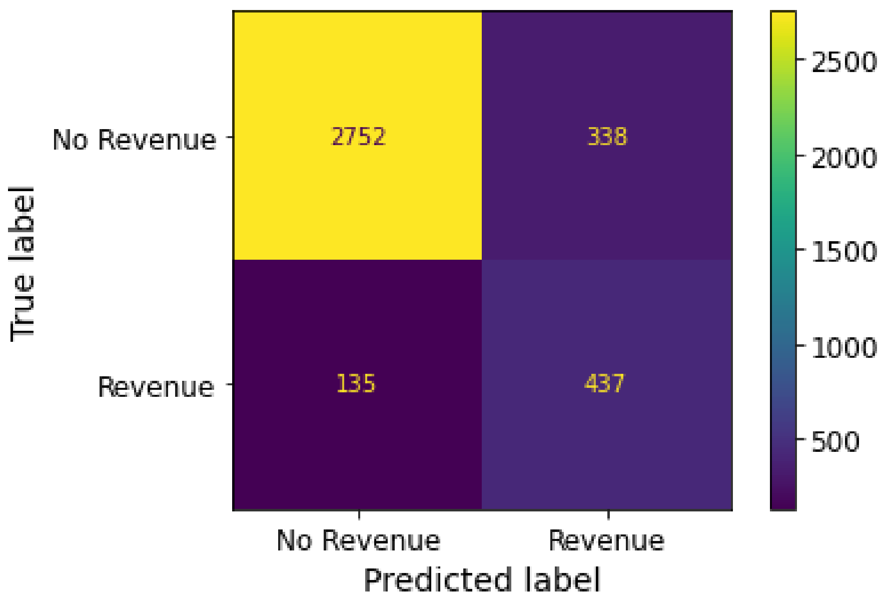
2.4 SVM

Support Vector Machines (SVM) can be tuned for imbalanced data, and we can also check how various kernels behave.

```
from sklearn.svm import SVC

params = {
    "C": [1, 10, 50],
    "gamma": ["scale", 1, 0.1, 0.01],
    "kernel": ["linear", "rbf", "poly"],
    "degree": [2, 3, 4]
}

svm_clf = SVC(random_state=42, class_weight="balanced")
grid_svm_clf = GridSearchCV(svm_clf, params, scoring="balanced_accuracy")
grid_svm_clf.fit(X_train, y_train)
```

```
evaluate_grid(grid_svm_clf)
```

```
Best parameters:  {'C': 50, 'degree': 2, 'gamma': 'scale', 'kernel': 'rbf'}
Accuracy of best (means of cross-validated scores on train set):
0.8285570898617853
Accuracy of best (on test set):  0.8273004503587027
```



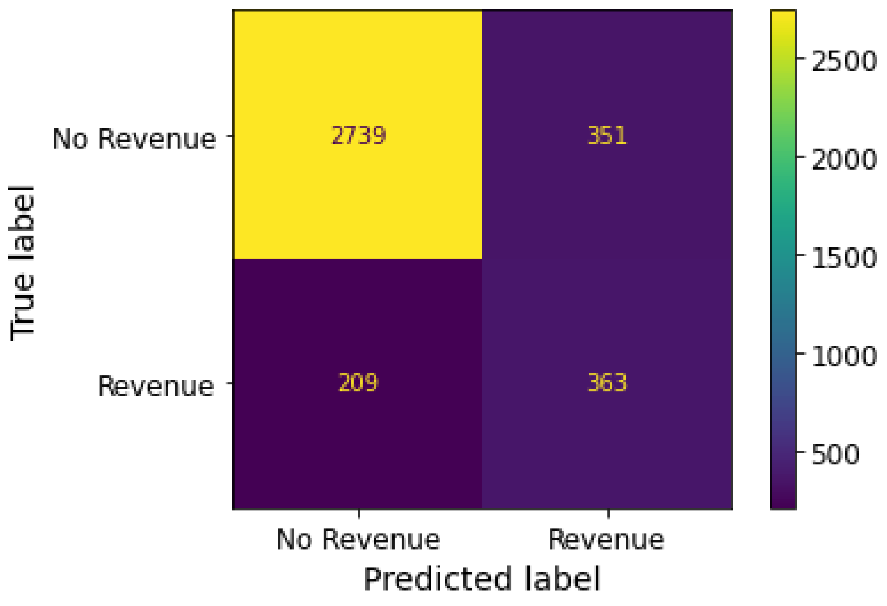
The resulting accuracy of the best model is high, but it's worth noticing that a considerable amount of time is required for training and cross-validation.

2.5 Naive Bayes

```
from sklearn.naive_bayes import GaussianNB


params = {
    "var_smoothing": np.logspace(0, -9, num=300)
}


gnb_clf = GaussianNB()
grid_gnb_clf = GridSearchCV(gnb_clf, params, scoring="balanced_accuracy")
grid_gnb_clf.fit(X_train, y_train)
```

```
evaluate_grid(grid_gnb_clf)
```

```
Best parameters:  {'var_smoothing': 0.00015040335536380253}
Accuracy of best (means of cross-validated scores on train set):
0.7703546558177674
Accuracy of best (on test set):  0.7605115758028379
```



3. Ensemble Classifiers

3.1 Bagging

The `BaggingClassifier` provided by scikit-learn allows to combine predictions from many base estimators. However we don't expect it to be particularly good for this data set, because it does not take in account the imbalance in the instances: the resulting classifier would have a bias towards the majority class.

We can verify if this assumption is true:

```
from sklearn.ensemble import BaggingClassifier

bag_clf = BaggingClassifier(random_state=42)
bag_clf.fit(X_train, y_train)
```

```
evaluate(bag_clf)
```

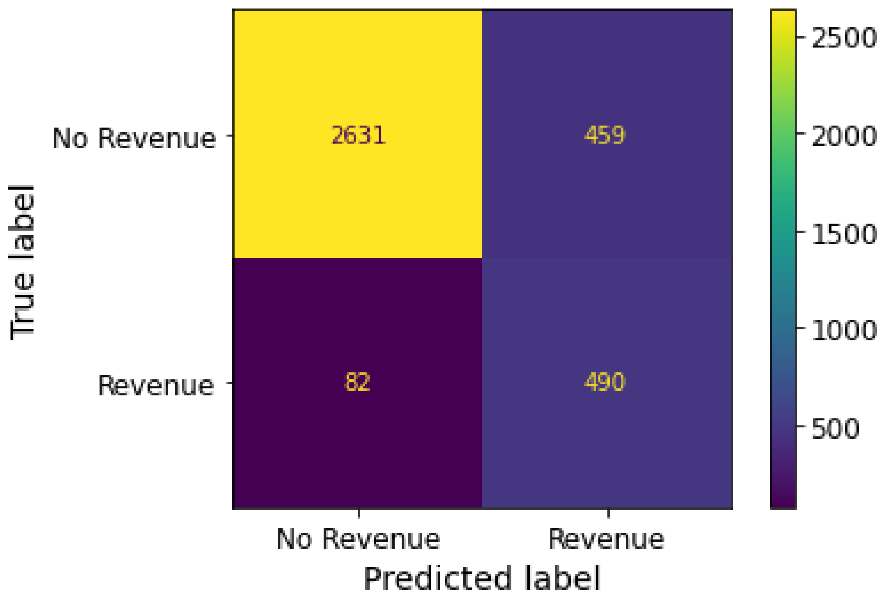
```
Accuracy (on test set):  0.745184103921968
```

The `BalancedBaggingClassifier` included in `imbalanced-learn` is more appropriate because it includes an additional step to balance the training set at fit time using a given sampler. The default sampler works by doing a random under-sampling of the majority class.

```
from imblearn.ensemble import BalancedBaggingClassifier

params = {
    "n_estimators": [100, 200, 500],
    "max_samples": [0.2, 0.5, 1.0],
    "max_features": [0.5, 1.0],
}

grid_bb_clf = GridSearchCV(
    BalancedBaggingClassifier(random_state=42),
    params,
    scoring="balanced_accuracy"
)


grid_bb_clf.fit(X_train, y_train)
```

```
evaluate_grid(grid_bb_clf)
```

```
Best parameters:  {'max_features': 1.0, 'max_samples': 0.2, 'n_estimators':␣
 ↪200}
Accuracy of best (means of cross-validated scores on train set):
0.8520401962169952
Accuracy of best (on test set):  0.8540498336614841
```



3.2 Random Forest

For the same reason as basic bagging, we can expect random forests and extra trees to produce unsatisfying results.

```
from sklearn.ensemble import RandomForestClassifier


rf_clf = RandomForestClassifier(random_state=42, class_weight="balanced")
rf_clf.fit(X_train, y_train)
```

```
evaluate(rf_clf)
```

```
Accuracy (on test set):  0.7634756828931586
```

```
from sklearn.ensemble import ExtraTreesClassifier

et_clf = ExtraTreesClassifier(random_state=42, class_weight="balanced")
et_clf.fit(X_train, y_train)
```

```
evaluate(et_clf)
```

```
Accuracy (on test set):  0.7611769298662503
```

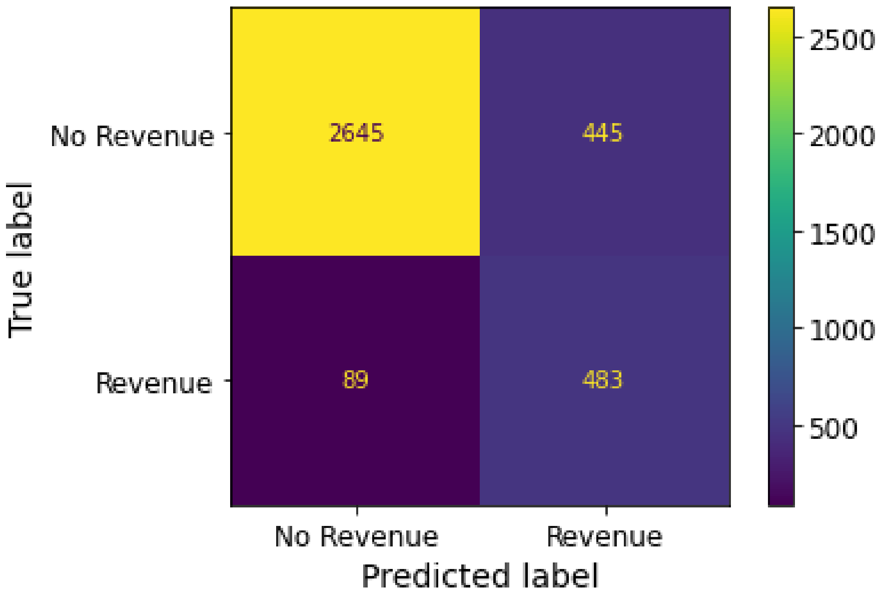
Both method have a relatively low (balanced) accuracy, even if the class weight is set to `balanced`.

`BalancedRandomForest` should provide better results, by under-sampling the majority class.

```
from imblearn.ensemble import BalancedRandomForestClassifier

params = {
    "n_estimators": [200, 300],
    "criterion": ["gini", "entropy"],
    "max_depth": list(range(5, 10)),
    "max_features": ["log2", "sqrt"],
}

grid_brf_clf = GridSearchCV(
    BalancedRandomForestClassifier(random_state=42, oob_score=True),
    params,
    scoring="balanced_accuracy"
)

grid_brf_clf.fit(X_train, y_train)
```

```
evaluate_grid(grid_brf_clf)
```

```
Best parameters:  {'criterion': 'gini', 'max_depth': 8, 'max_features':␣
 ↪'log2',
'n_estimators': 300}
Accuracy of best (means of cross-validated scores on train set):
0.8554375498184059
Accuracy of best (on test set):  0.8501963247108878
```



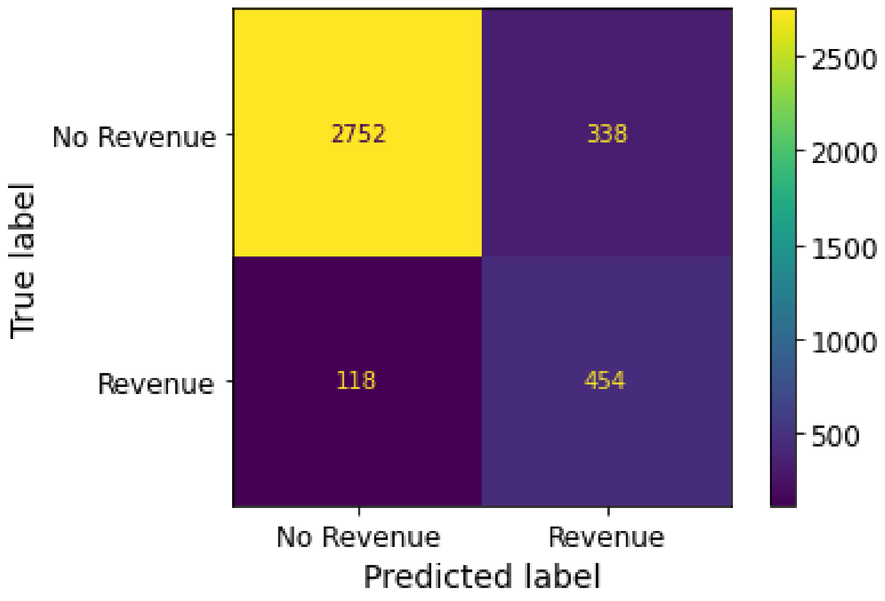
3.3 Random Undersampling with Boosting

`imbalanced-learn` also includes `RUSBoostClassifier`, which does random under-sampling integrated in the learning of AdaBoost.

```
from imblearn.ensemble import RUSBoostClassifier

params = {
    "n_estimators": [50, 100, 200],
    "learning_rate": [0.01, 0.1, 1],
}

grid_rus_clf = GridSearchCV(
    RUSBoostClassifier(random_state=42),
    params,
    scoring="balanced_accuracy"
)

grid_rus_clf.fit(X_train, y_train)
```

```
evaluate_grid(grid_rus_clf)
```

```
Best parameters:  {'learning_rate': 0.1, 'n_estimators': 50}
Accuracy of best (means of cross-validated scores on train set):
0.8451061106531841
Accuracy of best (on test set):  0.8421605902188427
```



3.4 EasyEnsemble

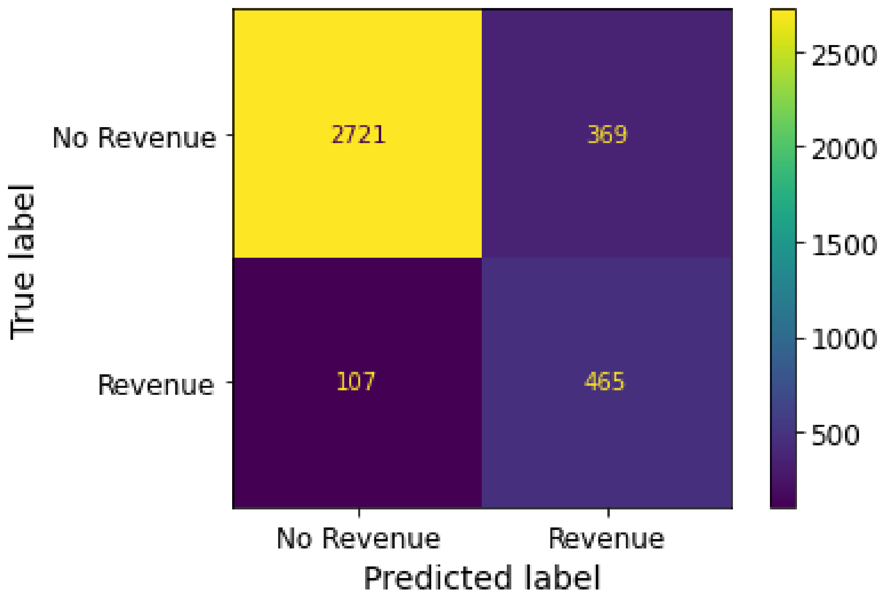
The final classifier included in `imbalanced-learn` is `EasyEnsemble`, that is an ensemble of AdaBoost learners trained on different balanced boostrap samples. The balancing is achieved by random under-sampling.

```
from imblearn.ensemble import EasyEnsembleClassifier
```

```
params = {
    "n_estimators": [50, 100, 200],
}

grid_ee_clf = GridSearchCV(
    EasyEnsembleClassifier(random_state=42),
    params,
    scoring="balanced_accuracy"
)

grid_ee_clf.fit(X_train, y_train)
```

```
evaluate_grid(grid_ee_clf)
```

```
Best parameters:  {'n_estimators': 200}
Accuracy of best (means of cross-validated scores on train set):
0.8456847750690647
Accuracy of best (on test set):  0.8467597936044537
```



3.5 XGBoost

XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting that solve many data science problems in a fast and accurate way.

To work with imbalanced data sets, it's recommended to set the `scale_pos_weight` to the ratio of negative instances to the positive ones.

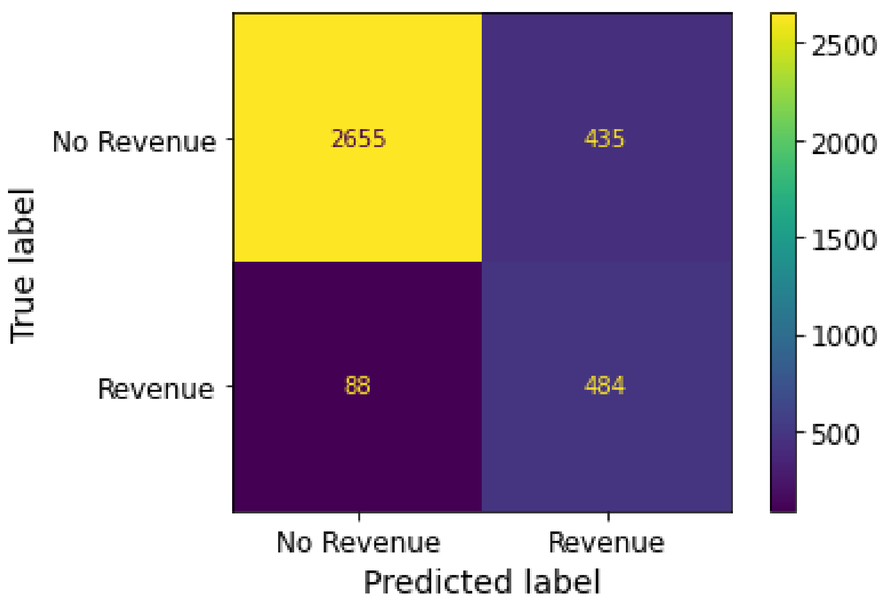
```
from collections import Counter

counter = Counter(y)
estimate = counter[0] / counter[1]
print("Estimate: %.3f" % estimate)
```

```
Estimate: 5.397
```

```
from xgboost import XGBClassifier

xgb_clf = XGBClassifier(random_state=42, scale_pos_weight=estimate,
                        objective="binary:logistic", verbosity=0)

params = {
    "n_estimators": [50, 100],
    "max_depth": [3, 5, 10],
    "learning_rate": [0.1, 0.01],
    "subsample": [0.8, 1],
    "colsample_bytree": [0.5, 0.8]
}

grid_xgb_clf = GridSearchCV(xgb_clf, params, scoring="balanced_accuracy")
grid_xgb_clf.fit(X_train, y_train)
```

```
evaluate_grid(grid_xgb_clf)
```

```
Best parameters:  {'colsample_bytree': 0.8, 'learning_rate': 0.01,␣
 ↪'max_depth':
5, 'n_estimators': 50, 'subsample': 0.8}
Accuracy of best (means of cross-validated scores on train set):
0.8556706675289127
Accuracy of best (on test set):  0.8526885735623599
```



3.6 Voting

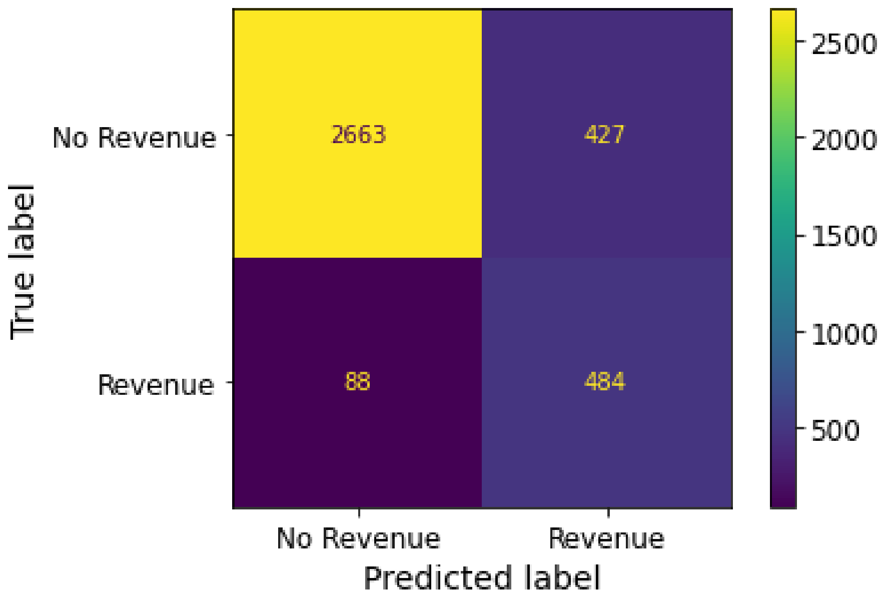
Finally, we can combine all the previous ensemble classifiers into a *Voting Ensemble*. We will begin with a hard voting ensemble, where each classifier has the same weight.

```python
from sklearn.ensemble import VotingClassifier


def get_models():
    models = list()
    models.append(("bb",  grid_bb_clf.best_estimator_))
    models.append(("brf",  grid_brf_clf.best_estimator_))
    models.append(("rus",  grid_rus_clf.best_estimator_))
    models.append(("ee",  grid_ee_clf.best_estimator_))
    models.append(("xgb",  grid_xgb_clf.best_estimator_))
    return models
```

```python
hard_voting_clf = VotingClassifier(estimators=get_models(), voting="hard")
hard_voting_clf.fit(X_train, y_train)
```

```python
evaluate(hard_voting_clf)
print_confusion_matrix(y_test, hard_voting_clf.predict(X_test))
```

```
Accuracy (on test set):  0.853983071944237
```



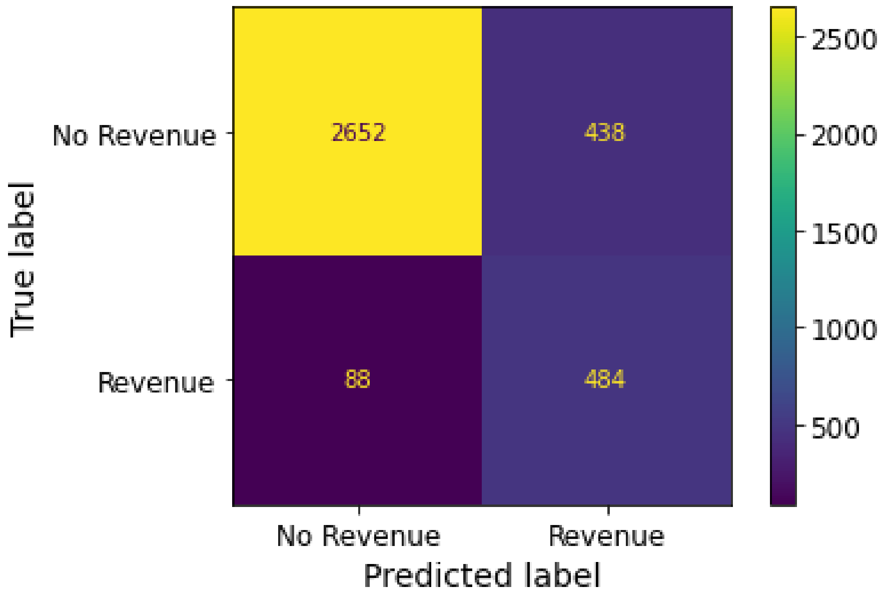
We can also create a soft voting classifier, where the weights of the classifiers are determined by the accuracy of the individual classifiers.

```python
# relative weights
model_scores = {
   name: balanced_accuracy_score(
      y_train,
      model.predict(X_train),
   )
   for name, model in get_models()
}
total_score = sum(model_scores.values())
```

```
soft_voting_clf = VotingClassifier(estimators=get_models(), voting="soft", ␣
 ↪weights=[
    model_scores[name] / total_score
    for name, _ in get_models()
  ])
soft_voting_clf.fit(X_train, y_train)
```

```
evaluate(soft_voting_clf)
print_confusion_matrix(y_test, soft_voting_clf.predict(X_test))
```

```
Accuracy (on test set):  0.852203136669156
```

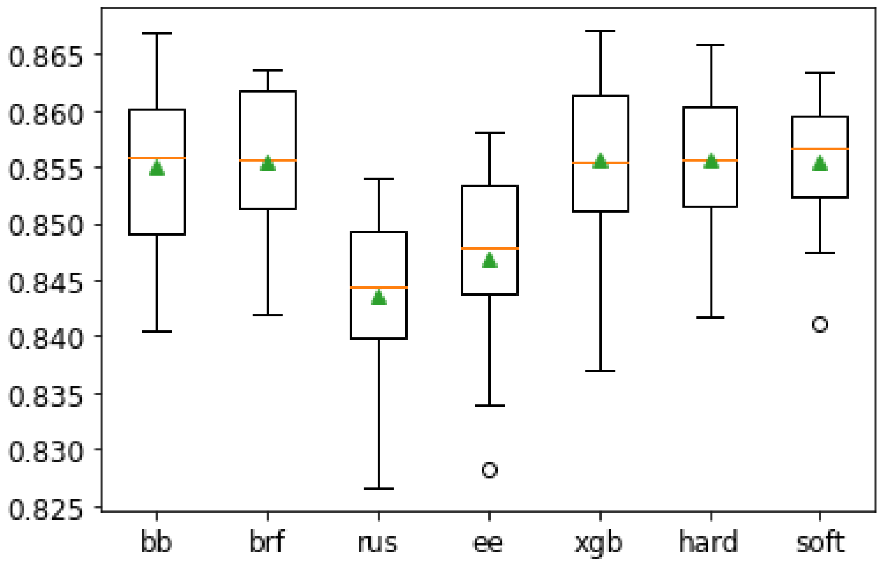


The hard voting ensemble has better results.

To estimate the effectiveness of the voting ensemble we can cross validate the single models, alongside the voting ensemble. Then we use a boxplot to show the accuracy of each cross-validated model.

```
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.model_selection import cross_val_score


def evaluate_model(model):
    cv = RepeatedStratifiedKFold(n_splits=5, n_repeats=3, random_state=42)
    scores = cross_val_score(
        model, X, y, scoring="balanced_accuracy", cv=cv, n_jobs=-1,␣
 ↪error_score="raise")
    return scores

eval_list = get_models()
eval_list.append(("hard", hard_voting_clf))
eval_list.append(("soft", soft_voting_clf))

results, names = list(), list()
for name, model in eval_list:
    scores = evaluate_model(model)
    results.append(scores)
    names.append(name)
```

```
plt.boxplot(results, labels=names, showmeans=True)
plt.show()
```



The voting ensembles have a slightly lower accuracy on the test set than the xgb classifier, but they do have a considerably lower variance.

4. Neural Network

To easily create a basic Neural Network, `keras` can be used. To test the effectiveness of a Neural Network to this particular problem, we create a baseline neural network with one hidden layer of 10 nodes.

```python
from tensorflow import keras

model = keras.models.Sequential([
    # The number of inputs has to be equal
    # to the number of features of the dataset
    keras.Input(shape=X_train.shape[1]),
    # Hidden layer
    keras.layers.Dense(10, activation="relu"),
    # One output node, which is a sigmoid:
    # the value will be continuous between 0 and 1.
    # To cast it to binary, we will need to assign
    # value 1 if greater than 0.5, or else 0.
    keras.layers.Dense(1, activation="sigmoid")
])
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 10)                80
 dense_1 (Dense)             (None, 1)                 11
=================================================================
Total params: 91
Trainable params: 91
Non-trainable params: 0
_________________________________________________________________
```

Now that the neural network has been defined, it needs to be compiled. While it is possible to use a precompiled model rather than compiling a new one. Since the model is simple and requires very little computation time, we will build it from scratch.

The same document suggests `Adam` as a sensible optimizer for the network.

Since the output needs to be binary, `BinaryCrossentropy()` is an appropriate loss function.

We also want to use the balanced accuracy to measure the performance of the neural network. Since it's not defined in the Keras library, we will need to define it ourselves.

```
class BalancedBinaryAccuracy(keras.metrics.BinaryAccuracy):
    def __init__(self, name='balanced_binary_accuracy', dtype=None):
        super().__init__(name, dtype=dtype)

    def update_state(self, y_true, y_pred, sample_weight=None):
        y_flat = y_true
        if y_true.shape.ndims == y_pred.shape.ndims:
            y_flat = tf.squeeze(y_flat, axis=[-1])
        y_true_int = tf.cast(y_flat, tf.int32)

        cls_counts = tf.math.bincount(y_true_int)
        cls_counts = tf.math.reciprocal_no_nan(tf.cast(cls_counts, self.
 ↪dtype))
        weight = tf.gather(cls_counts, y_true_int)
        return super().update_state(y_true, y_pred, sample_weight=weight)

model.compile(
    optimizer="adam",
    loss=keras.losses.BinaryCrossentropy(),
    metrics=[BalancedBinaryAccuracy()]
)
```

To validate the performance of the neural network we use `train_test_split` to generate a validation set, which is composed by the 20% of the training set. The original testing set is used as is.

```
X_test_nn, y_test_nn = X_test, y_test


X_train_nn, X_val_nn, y_train_nn,  y_val_nn = train_test_split(
    X_train, y_train, test_size=0.2, stratify=y_train, random_state=42)
```

The compiled model can be now trained: as a test, we will run it with 10 epochs. This number is, once again, arbitrary and will most likely need to be tweaked.

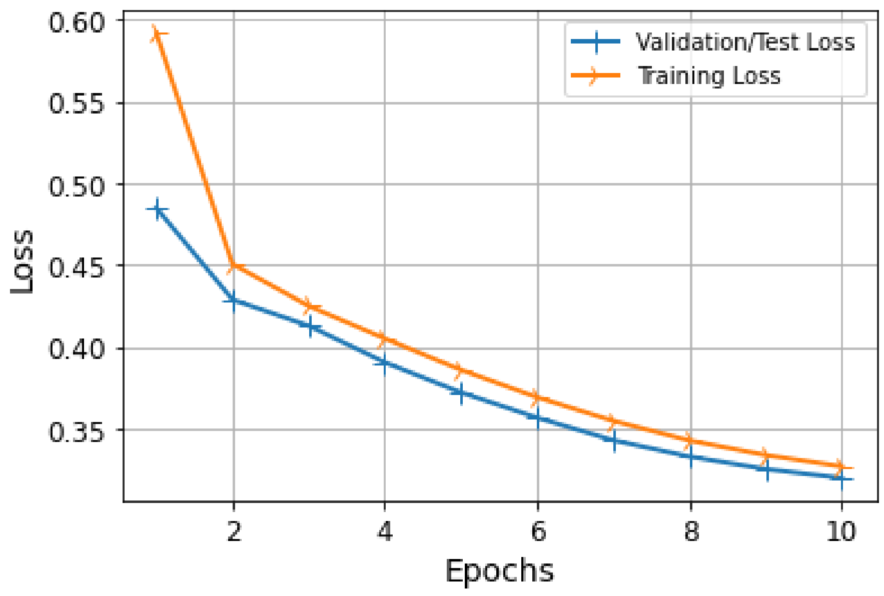
The `y_train` list cannot however be used directly, because the neural network cannot directly output a categorical value: to overcome this, the `y_train` list is transformed using Hot-Encoding.

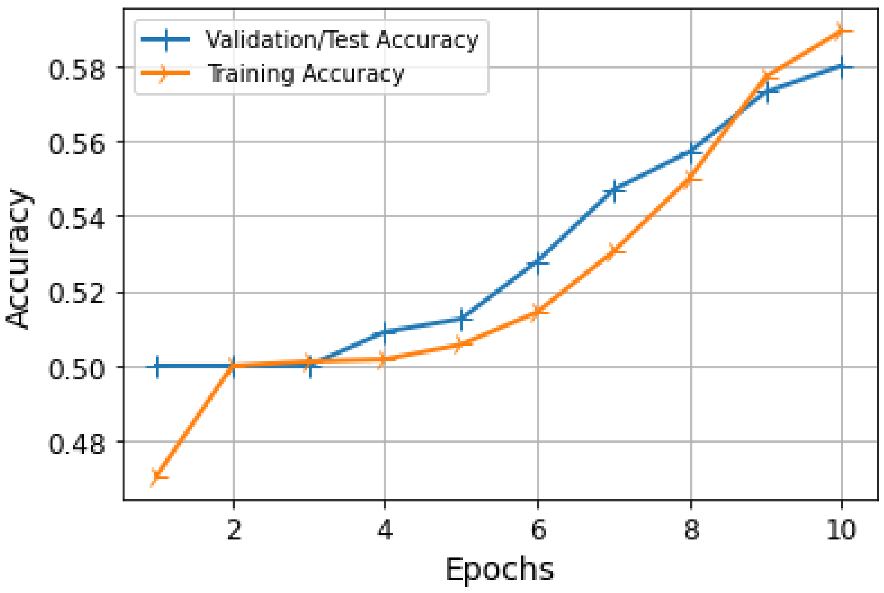
```
history = model.fit(
    X_train_nn,
    y_train_nn,
    epochs=10,
    validation_data=(X_val_nn, y_val_nn))
```

To evaluate the model we can begin by looking at evolution of the value of the loss function.

```
def plot_results(history):
    """
    Plots the results of the model training

    :param history: object returned by the train function that contains all␣
 ↪train infos
    """
    history_dict = history.history

    loss_values = history_dict['loss']
    val_loss_values = history_dict['val_loss']

    epochs = range(1, len(loss_values) + 1)

    # Plot line charts for both Validation and Training Loss
    line1 = plt.plot(epochs, val_loss_values, label='Validation/Test Loss')
    line2 = plt.plot(epochs, loss_values, label='Training Loss')
    plt.setp(line1, linewidth=2.0, marker='+', markersize=10.0)
    plt.setp(line2, linewidth=2.0, marker='4', markersize=10.0)
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.grid(True)
    plt.legend()
    plt.show()

    history_dict = history.history

    acc_values = history_dict['balanced_binary_accuracy']
    val_acc_values = history_dict['val_balanced_binary_accuracy']

    epochs = range(1, len(loss_values) + 1)

    line1 = plt.plot(epochs, val_acc_values, label='Validation/Test␣
 ↪Accuracy')
    line2 = plt.plot(epochs, acc_values, label='Training Accuracy')
    plt.setp(line1, linewidth=2.0, marker='+', markersize=10.0)
    plt.setp(line2, linewidth=2.0, marker='4', markersize=10.0)
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.grid(True)
    plt.legend()
    plt.show()
```

```
plot_results(history)
```





The graph shows that the loss function, at 10 epochs, is still decreasing: a higher number of epochs, with the same model configuration, should lower the loss function value, and improve the quality of the predictions.

A too high number of epochs however may lead to overfitting.

We need to check the accuracy on the test set:

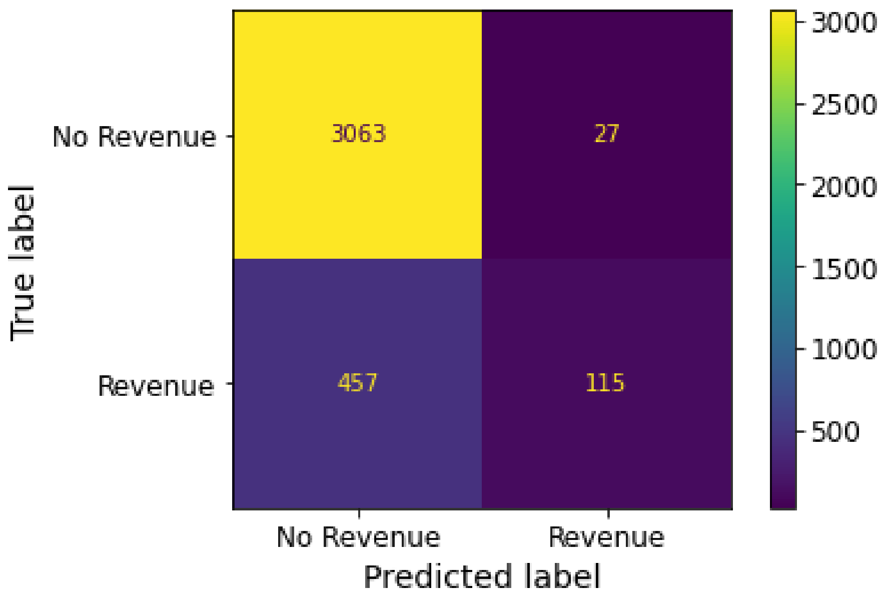
```
y_pred_nn = model.predict(X_test_nn).flatten()
y_pred_nn = pd.Series(y_pred_nn).map(lambda y: 1 if y >= 0.5 else 0)


print("Accuracy: ", balanced_accuracy_score(y_test_nn, y_pred_nn))
```

```
Accuracy:  0.5961555434856406
```

We can see that the accuracy is quite low. Parameter tuning may allow to find a neural network with higher accuracy.

We can also plot a heat map of the confusion matrix of the model

```
from sklearn.metrics import ConfusionMatrixDisplay

y_pred = model.predict(X_test).flatten()
y_pred = pd.Series(y_pred).map(lambda y: 1 if y >= 0.5 else 0)

print_confusion_matrix(y_test, y_pred)
```



Starting from this basic neural network we also gradually tried increasing the number of layers, and nodes per layer. We also tried adding Dropout layers. The best model that we found with this manual process has an accuracy of almost 78

4.1 Hyper-parameter Tuning

To tune the hyper parameters of a Keras neural network, we can use the `keras-tuner` package.

```
from keras import layers
import keras_tuner
```

```python
def build_model(hp):
    # Initialize the Sequential API and start stacking the layers
    model = keras.Sequential()
    # Input layer
    model.add(layers.Flatten())
    # Select number of hidden layer, between 1 and 3
    for i in range(hp.Int("num_layers", 1, 3)):
        model.add(
            layers.Dense(
                # Tune number of units separately.
                # Choose an optimal value between 32-512
                units=hp.Int(f"units_{i}", min_value=32,
                             max_value=512, step=32),
                activation="relu"
            )
        )
        # Tune whether to use dropout at the end of this layer
        if hp.Boolean(f"dropout_{i}"):
            model.add(layers.Dropout(rate=0.25))
    # Output layer
    model.add(layers.Dense(1, activation="sigmoid"))
    # Define the optimizer learning rate as a hyperparameter.
    learning_rate = hp.Float("lr", min_value=1e-4,
                             max_value=1e-2, sampling="log")
    model.compile(optimizer=keras.optimizers.
 ↪Adam(learning_rate=learning_rate),
                  loss=keras.losses.BinaryCrossentropy(),
                  metrics=[BalancedBinaryAccuracy()])

    return model
```

Now we can run the search for the best parameters. We use the `hyperband` search algorithm, which randomly samples all the combinations of hyper-parameters and instead of running full training and evaluation on the models, it trains each model for a few epochs with these combinations and select the best candidates based on the results on these few epochs. It does this iteratively and finally runs full training and evaluation on the final chosen candidates.

```python
tuner = keras_tuner.Hyperband(
    hypermodel=build_model,
    objective=keras_tuner.Objective(
        "val_balanced_binary_accuracy", direction="max"),
    max_epochs=30,
    directory="mldm_nn",
    project_name="mldm_nn_hyperband",
)
```

```python
from IPython.display import clear_output


class ClearTrainingOutput(keras.callbacks.Callback):
    def on_train_end(*args, **kwargs):
        clear_output(wait=True)


tuner.search(
    X_train_nn,
    y_train_nn,
    epochs=100,
    shuffle=True,
    verbose=1,
    use_multiprocessing=True,
    workers=2,
    callbacks=[ClearTrainingOutput()],
    validation_data=(X_val_nn, y_val_nn)
)
```

```
val_balanced_binary_accuracy: 0.7370098829269409
```

```
Best val_balanced_binary_accuracy So Far: 0.8166695237159729
Total elapsed time: 00h 14m 17s
```

We can now see the optimal neural network configuration:

```
# Get the top model.
best_model = tuner.get_best_models(num_models=1)[0]

# Build the model.
# Needed for `Sequential` without specified `input_shape`.
best_model.build(input_shape=X.shape)
best_model.summary()
```

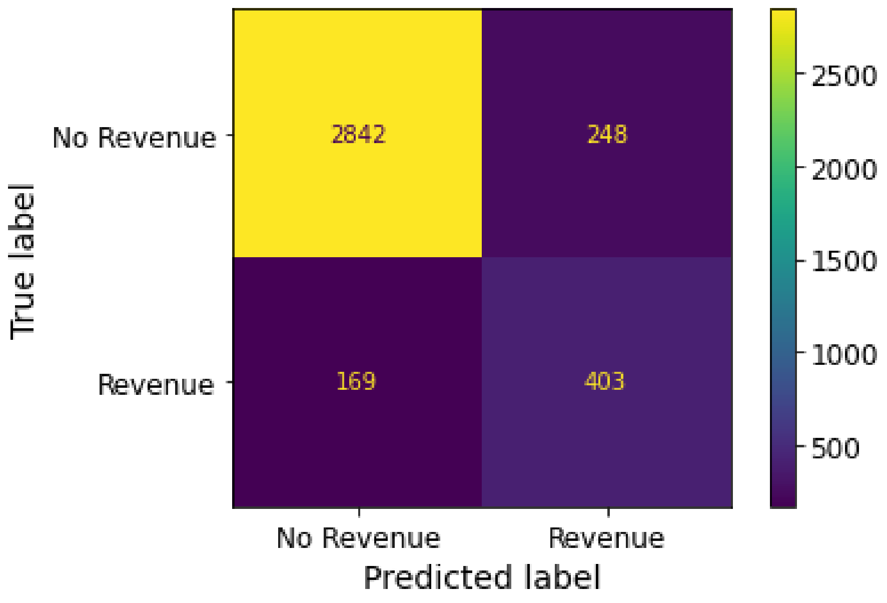
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (12205, 7)                0
 dense (Dense)               (12205, 320)              2560
 dropout (Dropout)           (12205, 320)              0
 dense_1 (Dense)             (12205, 64)               20544
 dropout_1 (Dropout)         (12205, 64)               0
 dense_2 (Dense)             (12205, 192)              12480
 dense_3 (Dense)             (12205, 1)                193
=================================================================
Total params: 35,777
Trainable params: 35,777
Non-trainable params: 0
_________________________________________________________________
```

```
y_pred = best_model.predict(X_test).flatten()
y_pred = pd.Series(y_pred).map(lambda y: 1 if y >= 0.5 else 0)

print("Accuracy: ", balanced_accuracy_score(y_test, y_pred))
```

```
Accuracy:  0.8121432774345396
```

The accuracy is not as high as the one obtained with the ensemble classifiers, but it's still quite high.

```
y_pred = best_model.predict(X_test).flatten()
y_pred = pd.Series(y_pred).map(lambda y: 1 if y >= 0.5 else 0)


print_confusion_matrix(y_test, y_pred)
```



5. Autosklearn

Auto-sklearn is an automated machine learning toolkit, which tries to identify the best possible model for a given data set. It is usually used as a starting point, from which the best model is improved manually. In this project however we will use Auto-sklearn to compare its results to ours.

We use Auto-sklearn 2.0, which at the time of writing is still in the experimental stage. It should anyway perform better than the previous version. We set 12 hours as the time limit for the execution, with a maximum of 30 minutes per run.

Once again, the models will be evaluated using balanced accuracy.

```
from autosklearn.experimental.askl2 import AutoSklearn2Classifier
from autosklearn.metrics import balanced_accuracy


auto_cls = AutoSklearn2Classifier(
    time_left_for_this_task=60*60*12,  # 12 hours
    per_run_time_limit=60*30,  # 30 minutes
    memory_limit=1024*3,  # 3 GB
    metric=balanced_accuracy,
    seed=42,
)
auto_cls.fit(X_train, y_train)
```

We can see the statistic of the Auto-sklearn execution using the `autosklearn` method:

```
print(auto_cls.sprint_statistics())
```

```
auto-sklearn results:
  Dataset name: 8c5908fd-1cbb-11ed-9451-3cf8629ba1c2
  Metric: balanced_accuracy
  Best validation score: 0.857500
  Number of target algorithm runs: 3578
  Number of successful target algorithm runs: 3567
  Number of crashed target algorithm runs: 11
  Number of target algorithms that exceeded the time limit: 0
  Number of target algorithms that exceeded the memory limit: 0
```

Using the `leaderboard` method we can see the ranking of the best models found.

To get the best performance out of the evaluated models, auto-sklearn builds an ensemble based on the models' prediction for the validation set. The `ensemble_weight` column represents the weight of the single model on the ensemble, `cost` the value of the loss function associated with the model, `duration` the length of time the model was optimized for.

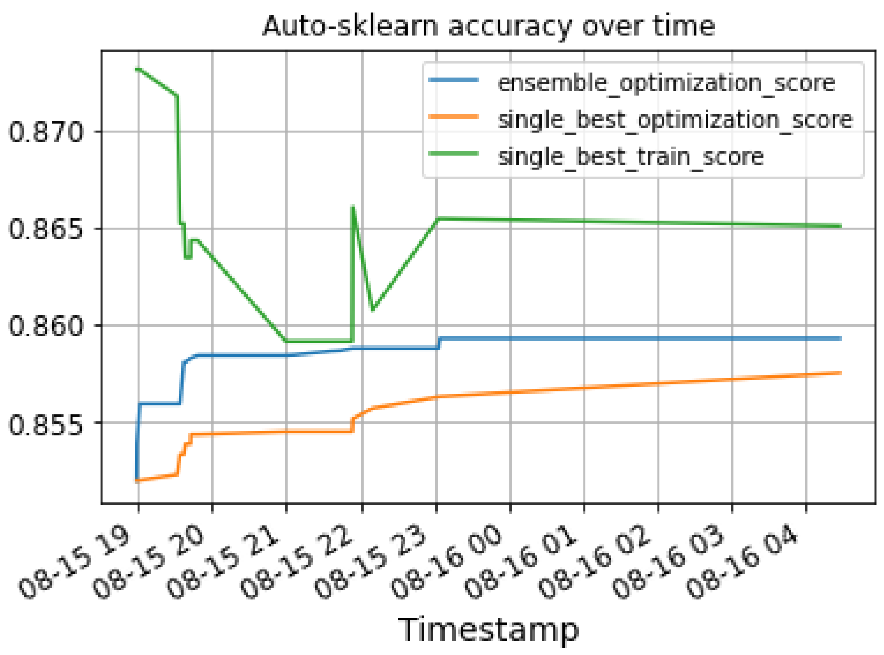
```
auto_cls.leaderboard()
```

```
model_id      rank  ensemble_weight               type       cost   duration
601             1             0.02  gradient_boosting  0.143332   9.012561
2441            2             0.04  gradient_boosting  0.144203   9.418808
3175            3             0.06  gradient_boosting  0.144219   8.163731
858             4             0.06  gradient_boosting  0.144732  15.188406
764             5             0.02  gradient_boosting  0.145272  11.095488
2081            6             0.02  gradient_boosting  0.145327   6.771889
396             7             0.04  gradient_boosting  0.145651   7.959959
533             8             0.02  gradient_boosting  0.145993   8.339468
358             9             0.02  gradient_boosting  0.147970  13.591912
736            10             0.06  gradient_boosting  0.148780  13.657828
212            11             0.02  gradient_boosting  0.150536  17.645834
2831           12             0.06  gradient_boosting  0.177535  14.373269
```

The best model(s) found by `autosklearn` are the ones using gradient boosting: this is consistent with our results, where we identified XGBoost (which is a variation of gradient boosting) as the most accurate model.

We can also plot the accuracy over time:

```
poT = auto_cls.performance_over_time_
poT.plot(
    x='Timestamp',
    kind='line',
    legend=True,
    title='Auto-sklearn accuracy over time',
    grid=True,
)
plt.show()
```



Finally, we evaluate the accuracy of the best auto-sklearn model on the test set.

```
evaluate(auto_cls)
```

```
Accuracy (on test set):  0.859584832643085
```

As we can see, the accuracy is close to the one of the XGBoost classifier.

6. Conclusions

For the basic classifiers, the ones that are most accurate are Decision Trees and Support Vector Machines. This is consistent with the results reported in [2], which identify Decision Trees as suited for the classification of the data set. All of the methods we have reported needed to be adapted to the imbalanced in the classes, mostly by setting `class_weight` appropriately.

As for ensemble methods, the basic versions included in `sklearn` didn't behave particularly well. The ensembles provided by `imblearn` proved much more accurate. We have also verified that by combining the ensembles in a voting ensemble allows to reduce bias, while also keeping a high accuracy.

The neural network that we developed is quite basic, but it still gives good results. More advanced neural network models are likely to be more suited: in [2] an LSTM recurrent neural networks is used.

Finally, Auto-sklearn confirmed the quality of gradient boosting for this problem.

Bibliography

[1] C.O. Sakar and Y. Kastro. *Online Shoppers Purchasing Intention Dataset Data Set.* Aug. 2018. URL: `https://archive.ics.uci.edu/ml/datasets/Online+Shoppers+Purchasing+Intention+Dataset`.

[2] C.O. Sakar et al. "Real-time prediction of online shoppers' purchasing intention using multilayer perceptron and LSTM recurrent neural networks." In: *Neural Computing and Applications* 31 (Oct. 2019), pp. 6893–6908. DOI: `https://doi.org/10.1007/s00521-018-3523-0`.

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

[4] 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.html`.

[5] François Chollet et al. *Keras.* `https://keras.io`. 2015.

[6] Matthias Feurer et al. "Efficient and Robust Automated Machine Learning". In: *Advances in Neural Information Processing Systems 28 (2015).* 2015, pp. 2962–2970.

[7] Matthias Feurer et al. "Auto-Sklearn 2.0: Hands-free AutoML via Meta-Learning". In: *arXiv:2007.04074 [cs.LG]* (2020).