Supervised binary classification on heart disease

Loïc Rakotoson

Abstract

In this paper we seek to perform a supervised classification to predict the presence of heart disease based on several patient measurements.
In the following, the classification models will be optimized with respect to the F1 score and Recall. This choice is explained by the counterparts of bad predictions. Indeed, it would be more tolerated to have false positives than undetected sick patients. With equal F1 performance, the Recall will allow to decide between two algorithms.

Introduction

When importing the data, the types given in the documentation were kept. However, some numerical variables were directly discretized based on their description in the same documentation.
Below are the first lines of our data.

```
data = (
    pd.read_csv(
        'heart.dat', header=None, sep='\s+', engine='python',
        dtype={
            0: np.int, 1: 'category', 2: 'category',
            3: np.int, 4: np.float, 5: np.bool,
            6: 'category', 7: np.int, 8: np.bool,
            9: np.float, 10: 'category', 11: 'category',
            12: 'category', 13: 'category'
        }
    ).rename(columns={
        0: "age", 1: "sex", 2: "chest_pain",
        3: "blood_pressure", 4: "cholesterol",
        5: "blood_sugar", 6: "electrocard",
        7: "heart_rate", 8: "angina", 9: "oldpeak",
        10: "slope_peak", 11: "vessels", 12: "thal",
        13: "target"
    })
)
```

Exploratory data analysis

To perform the analysis, groups of columns were created according to their type.

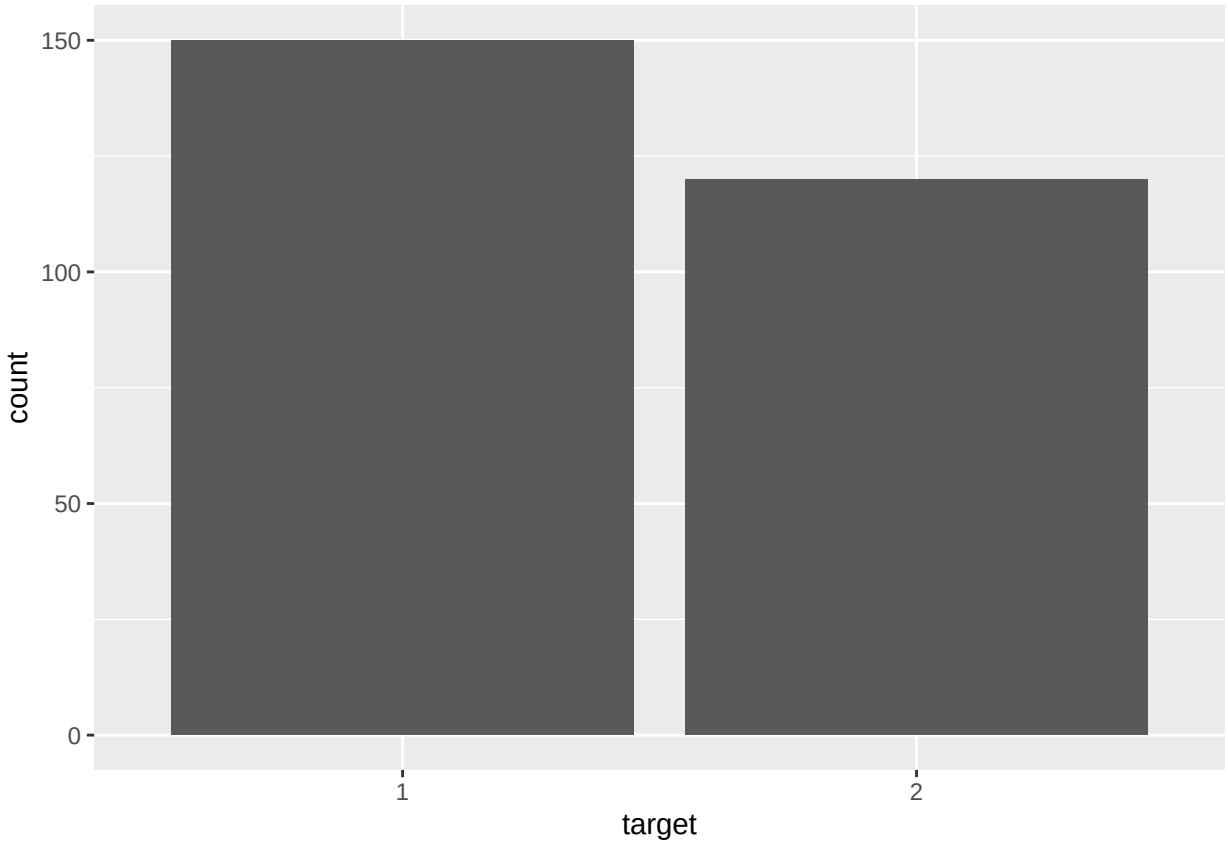
Table 1: First rows of data

age	sex	chest_pain	blood_pressure	cholesterol	blood_sugar	electrocard	heart_rate	angina	oldpeak	slope_peak	vessels	thal	target
70	1.0	4.0	130	322	FALSE	2.0	109	FALSE	2.4	2.0	3.0	3.0	2
67	0.0	3.0	115	564	FALSE	2.0	160	FALSE	1.6	2.0	0.0	7.0	1
57	1.0	2.0	124	261	FALSE	0.0	141	FALSE	0.3	1.0	0.0	7.0	2
64	1.0	4.0	128	263	FALSE	0.0	105	TRUE	0.2	2.0	1.0	7.0	1
74	0.0	2.0	120	269	FALSE	2.0	121	TRUE	0.2	1.0	1.0	3.0	1
65	1.0	4.0	120	177	FALSE	0.0	140	FALSE	0.4	1.0	0.0	7.0	1

The purpose of this analysis is to understand the data and derive information, including distributions, correlations and covariances.

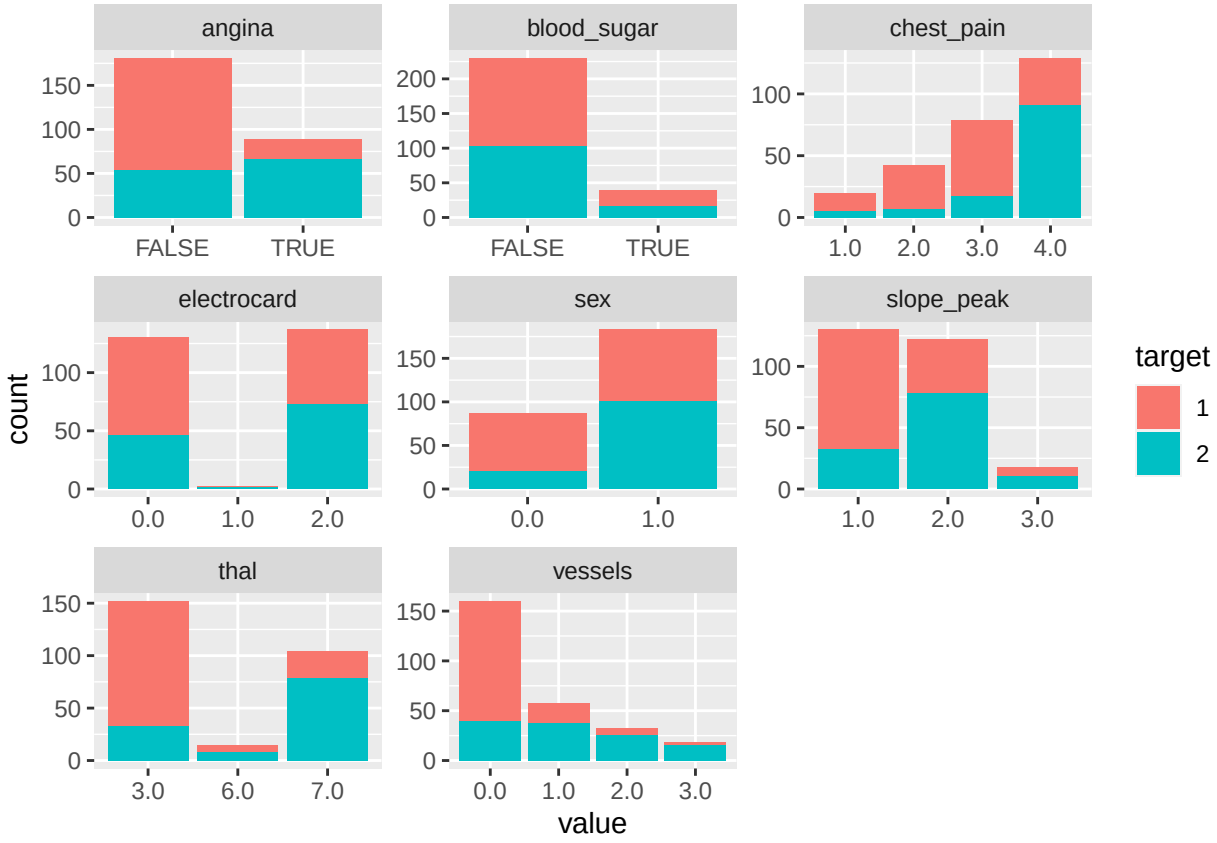
Target analysis

We have a binary target variable (`1`: absence and `2`: presence of heart disease).
We are not dealing with an unbalanced variable, so there will be no need to perform over-under-sampling strategies.



Analysis of qualitative variables

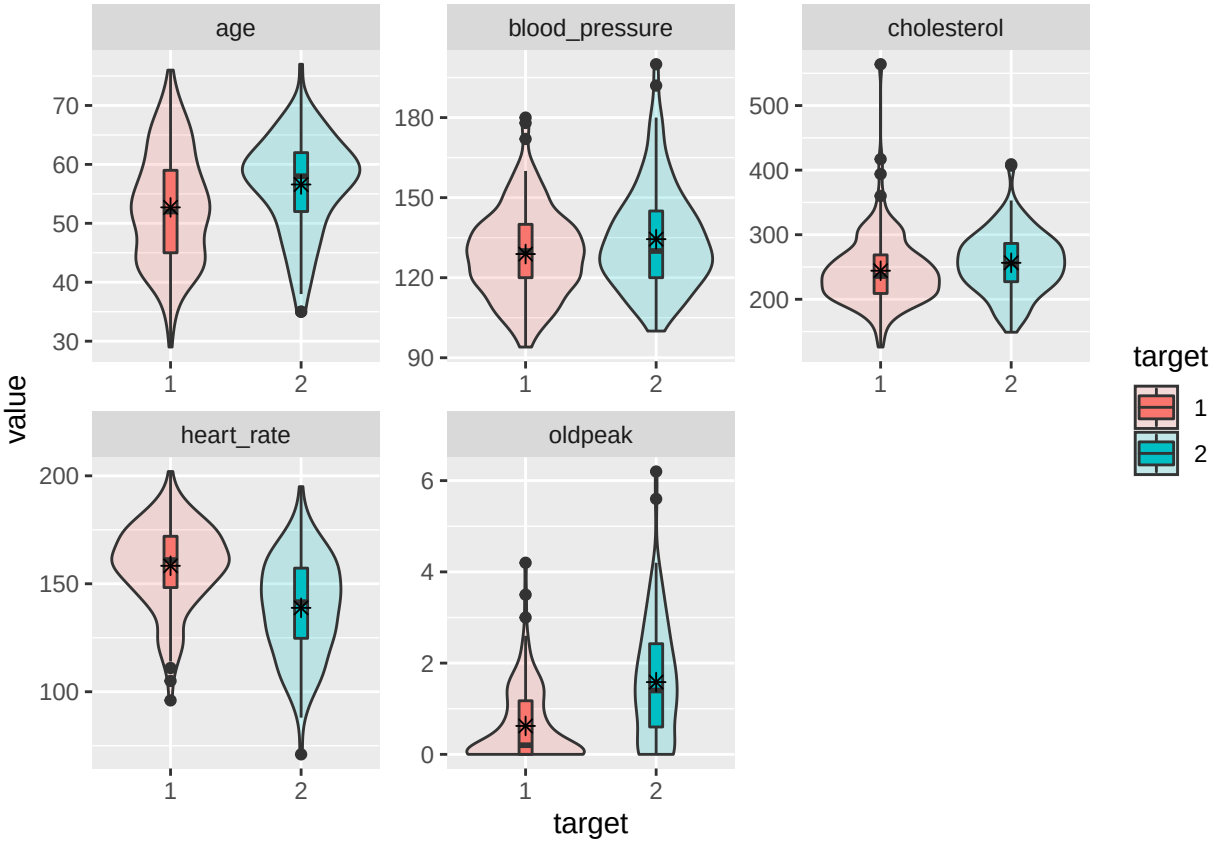
- Each modality of the variable `blood_sugar` has approximately the same target distribution. It does not vary the target.
- The distribution of the target in vessels is the same when it is present. It can be transformed into a binary variable of presence or not. We observe the same phenomenon with `chest_pain` where the `4` modality differs from the others which distribute the target in the same way.
- There are only 2 people with the `1` modality of the `electrocard` variable, one of which is positive to the target. This modality has no variance. On the other hand, the "0" mode seems to discriminate the target. If the electrocard variable is used, it can be converted into binary by imputing the modality `1` and assigning the individual to one of the other modalities by means of a closest neighbor algorithm or by an average method for example.



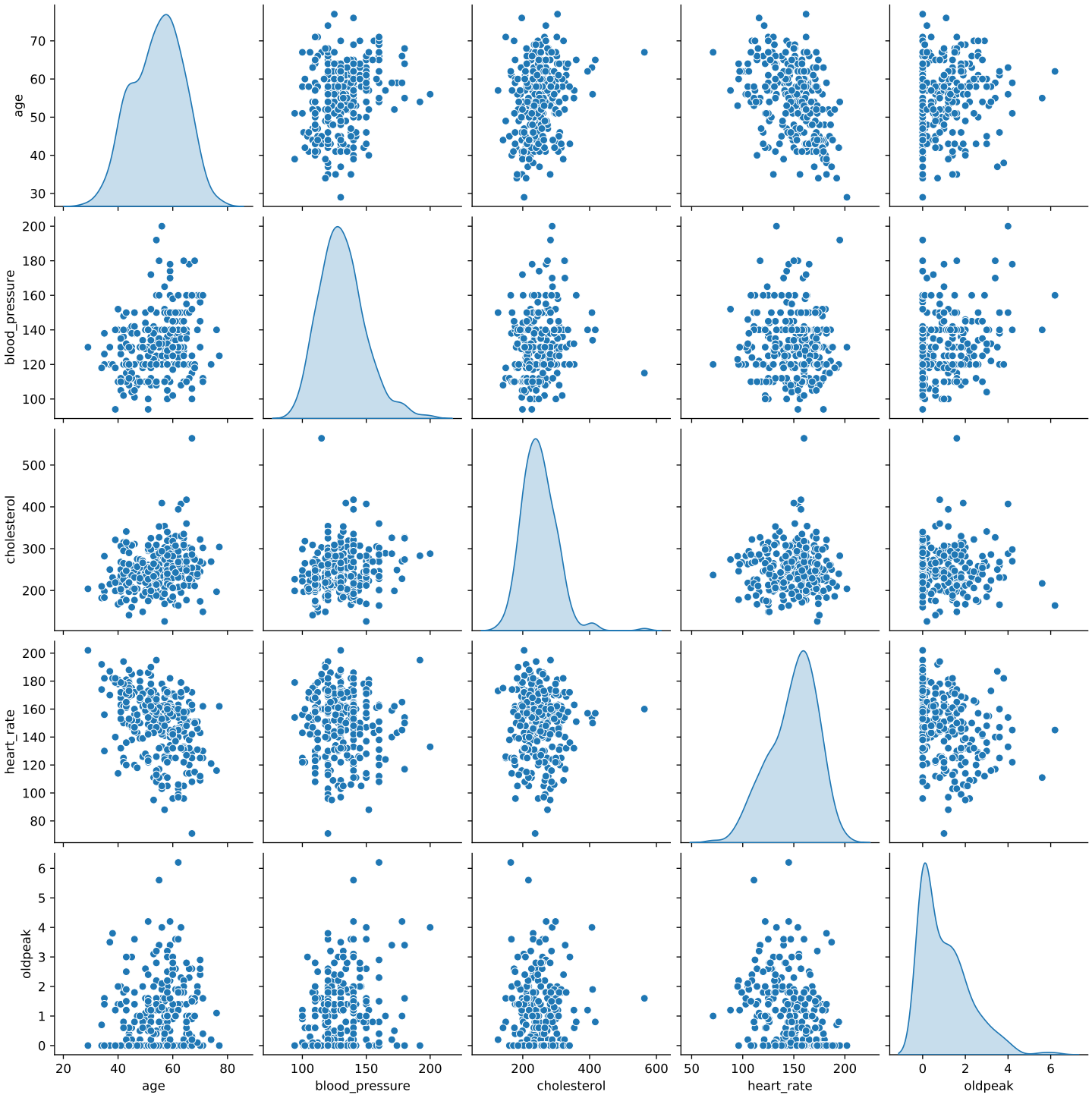
age	blood_pressure	cholesterol	heart_rate	oldpeak
Min. :29.00	Min. : 94.0	Min. :126.0	Min. : 71.0	Min. :0.00
1st Qu.:48.00	1st Qu.:120.0	1st Qu.:213.0	1st Qu.:133.0	1st Qu.:0.00
Median :55.00	Median :130.0	Median :245.0	Median :153.5	Median :0.80
Mean :54.43	Mean :131.3	Mean :249.7	Mean :149.7	Mean :1.05
3rd Qu.:61.00	3rd Qu.:140.0	3rd Qu.:280.0	3rd Qu.:166.0	3rd Qu.:1.60
Max. :77.00	Max. :200.0	Max. :564.0	Max. :202.0	Max. :6.20

Analysis of quantitative variables

- The `blood_presure` and `cholesterol` variables are each distributed approximately the same way with respect to the target, but with a slight shift.
- The `age`, `heart_rate` and `oldpeak` have a different distribution for each modality of the target.
- The variable 'oldpeak' has several outliers and there is a significant difference between the means and the medians for each of the target modalities. It is preferable to focus this variable on the median and reduce it between the 1st and 3rd quartile.


age
blood_pressure
cholesterol
heart_rate
oldpeak
value
target
target
1
2

The graph below shows the variable-variable relationships.
Note a certain negative correlation between the variable `age` and `heart_rate`.



Preprocessing

Based on the exploratory analysis of the data, the pre-processing that we will perform is as follows:

- Electrocard: Assigning the mode "1" to one of the other two with a "KNNImputer" and binarization.
- `vessels` and `chest_pain`: Grouping of variables according to the way the target is distributed and binarization
- oldpeak: standardization with respect to the median
- quantitative variables: standardization with respect to the mean

Table 2: Preprocessed data

inp__electrocard	topbin__vessels	topbin__chest_pain	bin__angina	bin__blood_sugar	bin__sex	ordinal__slope_peak	onehot__x0_6.0	onehot__x0_7.0	standard__age	standard__blood_pressure	standard__cholesterol	standard__heart_rate	scaler__oldpeak	target
1	0	0	1	0	1	0	0	0	-0.3986754	-0.3197880	-0.7046617	-1.0492113	0.3750	0
1	0	1	1	0	1	1	0	0	1.3454033	-1.7576357	0.8955578	-1.0492113	0.0625	1
1	0	1	0	0	1	0	0	0	-1.1617098	-1.1824966	-1.0023769	1.2032240	-0.5000	1
1	0	0	0	1	0	0	0	0	0.3643590	0.3128650	1.2677018	0.1203224	-0.5000	1
0	1	0	0	0	1	2	0	1	-0.7256901	-1.1824966	-0.4069464	0.8133794	0.1250	1
0	1	0	0	0	1	0	0	0	0.1463492	-0.6073575	-0.2766960	1.2465400	0.0000	0

Table 3: Naive benchmark results

model	accuracy	f1_score	recall
Logistic Regression	0.9074074	0.8979592	0.9166667
SGD Classifier	0.9074074	0.8936170	0.8750000
Neural Net	0.8888889	0.8800000	0.9166667
Naive Bayes	0.8888889	0.8800000	0.9166667
KNN	0.8888889	0.8695652	0.8333333
Gaussian Process	0.8703704	0.8571429	0.8750000
Random Forest	0.8703704	0.8510638	0.8333333
QDA	0.8703704	0.8510638	0.8333333
AdaBoost	0.8333333	0.8235294	0.8750000
Linear SVM	0.8518519	0.8181818	0.7500000
Decision Tree	0.8518519	0.8181818	0.7500000
RBF SVM	0.5740741	0.0800000	0.0416667

```
electropipe = Pipeline([('ls', LessFrequent()), ('scale', MinMaxScaler()),
                        ('inputer', KNNImputer()),
                        ('calib', Binarizer(threshold=.5))])

preprocess = ColumnTransformer(
    remainder='passthrough',
    transformers=[('inp', electropipe, ['electrocard']),
                  ('topbin', IsTop(), ['vessels', 'chest_pain']),
                  ('bin', Binarizer(), ['angina', 'blood_sugar', 'sex']),
                  ('ordinal', OrdinalEncoder(), ['slope_peak']),
                  ('onehot', OneHotEncoder(drop='first'), ['thal']),
                  ('standard', StandardScaler(), numerical[:-1]),
                  ('scaler', RobustScaler(), ['oldpeak'])])

label = LabelBinarizer()
```

The data is then separated into training and test sets.

```
x_train, x_test = train_test_split(data, test_size=.2, random_state=42069)
y_train = label.fit_transform(x_train.pop('target'))
y_test = label.transform(x_test.pop('target'))
```

Modeling

Naive Benchmark

In the following we will perform a naive benchmark of the classical models used for binary classification.
We will capture as score the f1, recall and accuracy.

We chose to optimize the first 3 algorithms as well as Adaboost which stands out with a good recall score despite the low f1.

Table 4: Best hyperparameters for logistic regression

preprocess__inp__inputer__n_neighbors	clf__penalty	clf__solver	clf__max_iter	name	f1_score	inner_score	recall	accuracy	count
3	l1	liblinear	100	Logistic Regression	0.7815603	0.8087427	0.8065476	0.8240741	3
3	l2	lbfgs	100	Logistic Regression	0.8279112	0.8152803	0.7426901	0.8518519	3

Optimization

For the optimization, we will choose the strategy of "Nested Grid Search Cross Validation" which consists in having 2 cross validations:

- A first one for the search of the best hyperparameters in a grid using a 4-fold KFold.
- A second one to obtain the score of the best hyperparameters chosen using a 6 Fold KFold, thus data not used for training.

We will also introduce the neighbor parameter to impute it even if with 2 individuals, the chance of its presence in a Fold is low.

```
cv_inner = KFold(n_splits = 4, shuffle = True, random_state = 42069)
cv_outer = KFold(n_splits = 6, shuffle = True, random_state = 42069)
```

Logistic Regression

As our dataset is quite light, it is interesting to see the absence or presence of regularization (l1 or l2).
The `liblinear` solver performs well on small datasets, while the 'lbfgs' solver theoretically handles well the absence of regularization.
Finally, the last parameter to be optimized is the number of iterations. We will choose the 3 "classical" values.

```
model1 = Pipeline([
    ('preprocess', preprocess),
    ('clf', LogisticRegression())
])

params = {
    'preprocess__inp__inputer__n_neighbors': np.arange(3, 10, 2),
    'clf__penalty': ['l1', 'l2', 'none'],
    'clf__solver': ['liblinear', 'lbfgs'],
    'clf__max_iter': [1e2, 1.5e2, 1e3]
}

result1, best1 = nestedCV(
    model1, params, x_train, y_train,
    cv_inner, cv_outer, name = "Logistic Regression",
)
```

Naive Bayes

The only hyperparameter to be optimized here is the variance share of the data for the variance update: `var_smoothing`.
We will choose a grid of 50 parameters between 1 and 10 -20 .

```
model2 = Pipeline([
    ('preprocess', preprocess),
    ('clf', GaussianNB())
])

params = {
    'preprocess__inp__inputer__n_neighbors': np.arange(3, 10, 2),
    'clf__var_smoothing': np.logspace(0, -20, num = 50)
}
```

Table 5: Naive Bayes best parameters

preprocess___inp___inputer___n_neighbors	clf___var_smoothing	name	f1_score	inner_score	recall	accuracy	count
3	0.0232995	Naive Bayes	0.8421053	0.8285947	0.8000000	0.8333333	1
3	0.0596362	Naive Bayes	0.7992342	0.8205153	0.7652778	0.8425926	3
3	0.1526418	Naive Bayes	0.8506944	0.7978685	0.8853383	0.8611111	2

```
result2, best2 = nestedCV(
    model2, params, x_train, y_train,
    cv_inner, cv_outer, name = "Naive Bayes"
)
```

Neural network

Multilayer perceptrons are optimized here.
The hyperparameters that we will optimize are:

- The size of the hidden layers
- The activation functions, `logistic` and `tanh` because our data is binary data, `relu` because all data is positive values.
- The optimizer, `sgd` classic, `adam` the `best` for now and `lbfgs` for small data sizes
- The behavior of α that decreases either as a function of the steps or as a function of the loss function.

We will activate the `early stopping` to avoid overfitting and we will allow ourselves to have a fairly high number of iterations.

```
model3 = Pipeline([
    ('preprocess', preprocess),
    ('clf', MLPClassifier(
        batch_size = 200,
        learning_rate_init = 1e-3,
        max_iter = 1e3,
        early_stopping = True,
        n_iter_no_change = 15
    ))
])

params = {
    'clf__hidden_layer_sizes': [
        (10,), (50,), (100,), (150,), (200,)],
    'clf__activation': ['logistic', 'relu', 'tanh'],
    'clf__solver': ['lbfgs', 'adam', 'sgd'],
    'clf__learning_rate': ['invscaling', 'adaptive']
}

result3, best3 = nestedCV(
    model3, params, x_train, y_train,
    cv_inner, cv_outer, name = "Neural Network"
)
```

Adaboost

For the Adaboost algorithm, we will choose the classical basic estimator, i.e. decission trees.
It is therefore also necessary to optimize their hyperparameters:

- The criterion for choosing the best pruning
- The pruning option

Table 6: Multilayer Perceptron best parameters

clf__hidden_layer_sizes	clf__activation	clf__solver	clf__learning_rate	name	f1_score	inner_score	recall	accuracy	count
10	logistic	lbfgs	adaptive	Neural Network	0.7200000	0.7621591	0.6000000	0.8055556	1
10	logistic	lbfgs	invscaling	Neural Network	0.7500000	0.7566822	0.7894737	0.7222222	1
50	logistic	lbfgs	invscaling	Neural Network	0.7096774	0.8075075	0.6875000	0.7500000	1
100	tanh	lbfgs	invscaling	Neural Network	0.6250000	0.7869252	0.5000000	0.6666667	1
200	logistic	lbfgs	invscaling	Neural Network	0.7142857	0.7883594	0.7142857	0.7777778	1
200	tanh	lbfgs	invscaling	Neural Network	0.5600000	0.7913585	0.5833333	0.6944444	1

Table 7: Adaboost best parameters

clf__base_estimator__criterion	clf__base_estimator__splitter	clf__n_estimators	name	f1_score	inner_score	recall	accuracy	count
entropy	random	10	Adaboost	0.7428571	0.7222410	0.6500000	0.7500000	1
entropy	random	50	Adaboost	0.7333333	0.7573710	0.7857143	0.7777778	1
gini	random	10	Adaboost	0.5833333	0.7296913	0.5833333	0.7222222	1
gini	random	50	Adaboost	0.8114618	0.7502782	0.8403509	0.8194444	2
gini	random	100	Adaboost	0.5714286	0.7789853	0.6250000	0.5833333	1

```
model4 = Pipeline([
    ('preprocess', preprocess),
    ('clf',
     AdaBoostClassifier(DecisionTreeClassifier(max_depth = None))
    )
])

params = {
    'clf__base_estimator__criterion': ["gini", "entropy"],
    'clf__base_estimator__splitter': ["best", "random"],
    'clf__n_estimators': [10, 50, 100, 200]
}

result4, best4 = nestedCV(
    model4, params, x_train, y_train,
    cv_inner, cv_outer, name = "Adaboost"
)
```

Evaluation

Finally, we perform the evaluation of each of the best models on the test set after they have been trained on the whole learning set.

It then emerges that logistic regression is the best model on the basis of f1 and recall.

Table 8: Models evaluation

model	accuracy	f1_score	recall
Logistic Regression	0.9074074	0.8979592	0.9166667
Naive Bayes	0.8703704	0.8444444	0.7916667
Adaboost	0.7777778	0.7391304	0.7083333
Neural Network	0.7592593	0.7111111	0.6666667

Appendix

```
class IsTop(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.top = list()

    def input(self, X):
        if isinstance(X, pd.DataFrame):
            X_ = X.to_numpy().copy()
        else: X_ = X.copy()
        return X_.astype(np.float).astype(np.int)

    def fit(self, X, y = None):
        X_ = self.input(X)
        for i in range(X_.shape[1]):
            self.top.append(np.bincount(X_[:,i]).argmax())

    def transform(self, X, y = None):
        X_ = self.input(X)
        for i in range(X_.shape[1]):
            X_[:,i] = X_[:,i] == self.top[i]
        X[X.columns] = X_
        return X

    def fit_transform(self, X, y = None):
        X_ = self.input(X)
        for i in range(X_.shape[1]):
            top = np.bincount(X_[:,i]).argmax()
            self.top.append(top)
            X_[:,i] = X_[:,i] == top
        X[X.columns] = X_
        return X
```

```
class LessFrequent(BaseEstimator, TransformerMixin):
    def __init__(self, missing = np.nan):
        self.missing = missing
        self.less = None

    def input(self, X):
        if isinstance(X, pd.DataFrame):
            X_ = X.to_numpy().copy().reshape((-1,))
        else: X_ = X.copy().reshape((-1,))
        return X_.astype(np.float).astype(np.int)

    def fit(self, X, y = None):
        X_ = self.input(X)
        self.less = np.bincount(X_).argmin()

    def transform(self, X, y = None):
        column = X.columns
        X_ = self.input(X)
        X = X.drop(column, axis = 1)
        X[column] = np.where(
            X_ == self.less, self.missing, X_
        ).astype(np.float).reshape((-1,1))
        return X

    def fit_transform(self, X, y = None):
        self.fit(X)
        return self.transform(X)
```

```
class Calibrate(BaseEstimator, TransformerMixin):
    def __init__(self, threshold):
        self.threshold = threshold

    def fit(self, X, y = None):
        return self

    def transform(self, X, y = None):
        return (X < self.threshold).astype(np.int)

    def fit_transform(self, X, y = None):
        return self.transform(X)
```

```
def gather( df, key, value, cols ):
    id_vars = [ col for col in df.columns if col not in cols ]
    id_values = cols
    var_name = key
    value_name = value
    return pd.melt( df, id_vars, id_values, var_name, value_name)
```

```
def nestedCV(estimator, param_grid, x_train, y_train, inner_cv, outer_cv,
             **kwargs):
    n_jobs = kwargs.get('n_jobs', -1)
    scoring = kwargs.get('scoring', 'f1')
    name = kwargs.get('name')
    results = list()
    x_train = x_train.reset_index(drop=True)

    for train, val in outer_cv.split(x_train, y_train):
        search = GridSearchCV(estimator,
                              param_grid,
                              cv=inner_cv,
                              scoring=scoring,
                              n_jobs=n_jobs,
                              refit=True).fit(x_train.iloc[train, :],
                                              y_train[train])
        y_pred = search.predict(x_train.iloc[val, :])
        result = search.best_params_
        result['name'] = name
        result['f1_score'] = f1_score(y_train[val], y_pred)
        result['inner_score'] = search.best_score_
        result['recall'] = recall_score(y_train[val], y_pred)
        result['accuracy'] = accuracy_score(y_train[val], y_pred)
        results.append(result)

    aggregate = pd.DataFrame(results).groupby(list(param_grid.keys()) +
                                              ['name'],
                                              as_index=False)
    table = aggregate.mean()
    table['count'] = aggregate.count().f1_score

    best_params = {
        k: v
        for k, v in table.sort_values(
            ['f1_score'], ascending=False).iloc[0].to_dict().items()
        if k in param_grid.keys()
    }

    return table, best_params
```