Exercise 9 - Ensemble Learning

First name: Brian

Last name: Schweigler

Matriculation number: 16-102-071

(1) Take the titanic dataset and using all attributes to predict the class ‘Survived’ (convert age and fare into classes ; exclude names from the attribute list) Build a boosting ensemble model with:

(a) Adaboost

(b) Gradientboost

(c) XGB

Show the Comparison of the Performance of the models

In [1]:
```
%load_ext autoreload
%autoreload 2
%matplotlib inline

import pandas as pd
import numpy as np
from sklearn import preprocessing
from mlxtend.evaluate import accuracy_score
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.model_selection import KFold
from sklearn.ensemble import GradientBoostingClassifier, AdaBoostClassifier, VotingClassifier
from xgboost.sklearn import XGBClassifier
from IPython.core.display import display
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier

df = pd.read_csv("data/titanic.csv", index_col='Name')
pd.set_option('display.max_colwidth', None)
le = preprocessing.LabelEncoder()
bins = [0, 4, 18, 65, 100]
labels = ['Infant', 'Child', 'Adult', 'Elderly']
labels = [1, 2, 3, 4]
df['AgeGroup'] = pd.cut(df['Age'], bins=bins, labels=labels, right=False)
df["AgeGroup"] = le.fit_transform(df["AgeGroup"])
df['FareGroup'] = pd.qcut(x=df['Fare'], q=4)
df["FareGroup"] = le.fit_transform(df["FareGroup"])
df["Survived"] = le.fit_transform(df["Survived"])
df["Sex"] = le.fit_transform(df["Sex"])
print(df.head(2))

models = {
    'AdaBoostClassifier': AdaBoostClassifier(),
    'Gradientboost': GradientBoostingClassifier(learning_rate=0.01),
    'XGBClassifier': XGBClassifier(),
}
```

```
                                                    Survived  Pclass  Sex  \
Name
Mr. Owen Harris Braund                                     0       3    1
Mrs. John Bradley (Florence Briggs Thayer) Cumings         1       1    0

                                                     Age  \
Name
Mr. Owen Harris Braund                              22.0
Mrs. John Bradley (Florence Briggs Thayer) Cumings  38.0

                                                    Siblings/Spouses Aboard  \
Name
Mr. Owen Harris Braund                                                    1
Mrs. John Bradley (Florence Briggs Thayer) Cumings                        1

                                                    Parents/Children Aboard  \
Name
Mr. Owen Harris Braund                                                    0
Mrs. John Bradley (Florence Briggs Thayer) Cumings                        0

                                                       Fare  AgeGroup  \
Name
Mr. Owen Harris Braund                               7.2500         2
Mrs. John Bradley (Florence Briggs Thayer) Cumings  71.2833         2

                                                    FareGroup
Name
```

```
Mr. Owen Harris Braund                                   0
Mrs. John Bradley (Florence Briggs Thayer) Cumings       3
```

In [2]:

```
all_features = ['Pclass', 'Sex', 'Siblings/Spouses Aboard',
                'Parents/Children Aboard', 'AgeGroup', 'FareGroup']


def kfold_boost_eval(model, x: pd.DataFrame, y: pd.DataFrame):
    kf = KFold(n_splits=5, shuffle=True, random_state=6)
    accuracy = np.empty(kf.n_splits)
    precision = np.empty(kf.n_splits)
    recall = np.empty(kf.n_splits)
    f1 = np.empty(kf.n_splits)

    i = 0
    for train_index, test_index in kf.split(x):
        x_train, x_test = x.iloc[train_index], x.iloc[test_index]
        y_train, y_test = y.iloc[train_index], y.iloc[test_index]

        model.fit(x_train, y_train)
        prediction = model.predict(x_test)

        accuracy[i] = accuracy_score(prediction, y_test)
        precision[i] = precision_score(prediction, y_test)
        recall[i] = recall_score(prediction, y_test)
        f1[i] = f1_score(prediction, y_test)
        i += 1

    print(name)
    return accuracy, precision, recall, f1


for name, model in models.items():
    a, p, r, f = kfold_boost_eval(model, df[all_features], df["Survived"])
    data = {
        'Fold': [1, 2, 3, 4, 5],
        'Accuracy': a,
        'Precision': p,
        'Recall': r,
        'F1-Score': f,
    }

    scores = pd.DataFrame(data).set_index('Fold')
    display(scores)
```

AdaBoostClassifier

Fold	Accuracy	Precision	Recall	F1-Score
1	0.820225	0.779412	0.757143	0.768116
2	0.764045	0.611111	0.758621	0.676923
3	0.779661	0.716418	0.705882	0.711111
4	0.762712	0.657143	0.718750	0.686567
5	0.813559	0.769231	0.735294	0.751880

Gradientboost

Fold	Accuracy	Precision	Recall	F1-Score
1	0.848315	0.632353	0.955556	0.761062
2	0.786517	0.541667	0.886364	0.672414
3	0.819209	0.656716	0.830189	0.733333
4	0.790960	0.542857	0.883721	0.672566
5	0.847458	0.646154	0.913043	0.756757

XGBClassifier

Fold	Accuracy	Precision	Recall	F1-Score
1	0.837079	0.691176	0.854545	0.764228
2	0.814607	0.611111	0.897959	0.727273
3	0.790960	0.776119	0.702703	0.737589

	Accuracy	Precision	Recall	F1-Score
Fold				
4	0.768362	0.671429	0.723077	0.696296
5	0.847458	0.769231	0.806452	0.787402

Seems like Gradientboost performs the best out of all 3.

(2) With your selected stock / market index using all attributes to predict 'daily returns'(decision). ('daily returns' must first be converted into a decision class that will be used as the target(label), all other attributes must be grouped into classes)

(a) Build a voting ensemble model and Explain how the voting technique affects the performance of the model. (b) Stack any models of your choice to create an ensemble. How does stacking compare with voting?

In [3]:

```python
from typing import Any

stock_df = pd.read_csv("data/Nasdaq.csv", index_col='Date')
print(stock_df.head(3))

daily_return = np.empty(stock_df['Close'].shape)
#  From Slides: Daily return (r): Difference in percentage between
#  price at time t+1 and time t
daily_return[0] = float('NaN')  # The first
daily_return[1:] = np.ediff1d(stock_df['Close']) / stock_df['Close'][:-1]
stock_df.insert(loc=len(stock_df.columns), column='Daily Return', value=daily_return)

binary = (daily_return > 0).astype(float)
stock_df.insert(loc=len(stock_df.columns), column='Binary Decision', value=binary)
stock_df["Binary Decision"] = le.fit_transform(stock_df["Binary Decision"])

stock_df['Rolling Mean 5'] = stock_df['Close'].rolling(5).mean()
stock_df['Rolling Mean 10'] = stock_df['Close'].rolling(10).mean()
stock_df['Rolling Mean 20'] = stock_df['Close'].rolling(20).mean()
stock_df['Rolling Mean 50'] = stock_df['Close'].rolling(50).mean()
stock_df['Rolling Mean 200'] = stock_df['Close'].rolling(200).mean()
stock_df = stock_df.fillna(0)  # NAs replaced with zero


def estimators() -> list[tuple[str, Any]]:
    return [
        ('KNN', KNeighborsClassifier(n_neighbors=3)),
        ('Tree', DecisionTreeClassifier()),
        ('Gaussian Bayes', GaussianNB())
    ]


# Rest is as you'd expect with fit and predict

print(stock_df.tail(2))
```

```
                  Open        High         Low       Close   Adj Close  Volume
Date
1971-02-05  100.000000  100.000000  100.000000  100.000000  100.000000       0
1971-02-08  100.839996  100.839996  100.839996  100.839996  100.839996       0
1971-02-09  100.760002  100.760002  100.760002  100.760002  100.760002       0
                    Open          High           Low         Close  \
Date
2021-09-20  14758.139648  14841.820312  14530.070312  14713.900391
2021-09-21  14803.400391  14847.027344  14696.467773  14779.216797

               Adj Close      Volume  Daily Return  Binary Decision  \
Date
2021-09-20  14713.900391  4860630000     -0.021940                0
2021-09-21  14779.216797  3083208000      0.004439                1

            Rolling Mean 5  Rolling Mean 10  Rolling Mean 20  Rolling Mean 50  \
Date
2021-09-20    15027.816016     15126.937012     15143.909961     14875.705195
2021-09-21    14976.107422     15067.425684     15135.738281     14876.624727

            Rolling Mean 200
Date
2021-09-20      13856.711787
2021-09-21      13868.721973
```

With the Rolling Mean, we already have a sort of grouping, thus I vouch to leave the values ungrouped and just work with the Rolling Mean, especially as it is what was also used in prior exercises.

In [4]:

```
stock_df["Class Daily Return"] = pd.qcut(stock_df["Daily Return"], q=3)
stock_df["Class Daily Return"], _ = stock_df["Class Daily Return"].factorize(sort=True)

all_stock_features = ['Volume', 'Rolling Mean 5', 'Rolling Mean 10',
                      'Rolling Mean 20', 'Rolling Mean 50', 'Rolling Mean 200']


def evaluate(model: Any, X_train: pd.DataFrame, X_test: pd.DataFrame, y_train: pd.DataFrame, y_test: pd.DataFrame,
             average: Any = 'binary', zero_division: Any = 'warn') -> (float, float, float, float):
    """
    Evaluates a model on a training and test set.
    :param model: The model to evaluate
    :param average: default='binary': This parameter is required for multiclass/multilabel targets.
    :param zero_division: default='warn'
    :return:
    """
    model.fit(X_train, y_train)
    pred = model.predict(X_test)

    accuracy = accuracy_score(pred, y_test)
    precision = precision_score(pred, y_test, average=average, zero_division=zero_division)
    recall = recall_score(pred, y_test, average=average, zero_division=zero_division)
    f1 = f1_score(pred, y_test, average=average, zero_division=zero_division)

    return accuracy, precision, recall, f1


def kfold_eval(model: Any, X: pd.DataFrame, y: pd.DataFrame, k: int, average: Any = 'binary',
               zero_division: Any = 'warn') -> (np.array, np.array, np.array, np.array):
    """
    Performs k-fold cross-validation on the given model.
    :param model: The model to evaluate
    :param X: the features to train on
    :param y: the labels to predict
    :param k: how many folds to perform
    :param average: default='binary': This parameter is required for multiclass/multilabel targets.
    :param zero_division: default='warn'
    :return: accuracy, precision, recall, f1
    """

    kf = KFold(n_splits=k)

    accuracy = np.empty(kf.n_splits)
    precision = np.empty(kf.n_splits)
    recall = np.empty(kf.n_splits)
    f1 = np.empty(kf.n_splits)

    i = 0
    for train_index, test_index in kf.split(X):
        accuracy[i], precision[i], recall[i], f1[i] = evaluate(
            model,
            X.iloc[train_index],
            X.iloc[test_index],
            y.iloc[train_index],
            y.iloc[test_index],
            average,
            zero_division,
        )
        i += 1

    return accuracy, precision, recall, f1


def boosting_performance(model: Any, target_feature: str = 'Daily Returns Class', k: int = 5, average: Any = 'binary',
                         zero_division: Any = 'warn') -> (np.array, np.array, np.array, np.array):
    """
    Performs a k-fold cross-validation on an SVM with the specified kernel.

    :param model: The SVM model
    :param target_feature: default='Daily Returns Class': The target feature
    :param k: how many folds to perform
    :param average: default='binary': This parameter is required for multiclass/multilabel targets.
    :param zero_division: default='warn'

    :return: accuracy, precision, recall, f1
    """
    X = stock_df[all_stock_features]
    y = stock_df["Class Daily Return"]

    return kfold_eval(model=model, X=X, y=y, k=k, average=average, zero_division=zero_division)
```

In [5]:

```python
k_folds = 5
for voting in ['soft', 'hard']:
    print(f'Voting = {voting}:')
    vote_model = VotingClassifier(estimators=estimators(), voting=voting)

    a, p, r, f = boosting_performance(vote_model, k=k_folds, average='macro', zero_division=0)
    data = {
        'Fold': range(1, k_folds + 1),
        'Accuracy': a,
        'Precision': p,
        'Recall': r,
        'F1-Score': f,
    }

    vote_scores = pd.DataFrame(data).set_index('Fold')
    display(vote_scores)
```

```
Voting = soft:
```

Fold	Accuracy	Precision	Recall	F1-Score
1	0.397420	0.341186	0.334484	0.243252
2	0.313180	0.336974	0.348685	0.254276
3	0.333871	0.313802	0.317459	0.307042
4	0.340726	0.313650	0.308428	0.302055
5	0.317339	0.327554	0.325754	0.261987

```
Voting = hard:
```

Fold	Accuracy	Precision	Recall	F1-Score
1	0.393793	0.334886	0.349451	0.229639
2	0.350262	0.330278	0.358524	0.286219
3	0.310081	0.323959	0.321672	0.305869
4	0.358468	0.329020	0.329986	0.314219
5	0.319355	0.333172	0.403910	0.210987

Hard voting entails picking the prediction with the highest number of votes, whereas soft voting entails combining the probabilities of each prediction in each model and picking the prediction with the highest total probability.

Performance is quite similar in this case here though.

(b) Stack any models of your choice to create an ensemble. How does stacking compare with voting?

In [6]:

```python
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import StackingClassifier

k_folds = 5
stack_model = StackingClassifier(estimators=estimators(), final_estimator=LogisticRegression(solver='saga', max_iter=5000)

a, p, r, f = boosting_performance(stack_model, k=k_folds, average='macro', zero_division=0)
data = {
    'Fold': range(1, k_folds+1),
    'Accuracy': a,
    'Precision': p,
    'Recall': r,
    'F1-Score': f,
}

stack_scores = pd.DataFrame(data).set_index('Fold')
stack_scores
```

Out[6]:

Fold	Accuracy	Precision	Recall	F1-Score
1	0.312374	0.330195	0.317820	0.204012
2	0.391778	0.337808	0.361225	0.278679
3	0.293952	0.350124	0.353973	0.245671

Fold	Accuracy	Precision	Recall	F1-Score
4	0.278226	0.342131	0.353248	0.220650
5	0.372984	0.364003	0.368794	0.295430

The stacking classifier is able to learn when our base estimators can be trusted or not. Stacking allows us to use the strength of each individual estimator by using their output as an input of a final estimator.

Performance seems to be slightly better with voting than with stacking in our case.