Rapport TD4

Sujet choisi : TD4 : Sélection de stocks à l'aide d'un arbre de décision

Récupérer le fichier dans python

Pour débuter, on récupère le fichier csv 'Eurostoxx50_EOD_Clean' représentant l'historique des séries financières des composantes de l'Eurostoxx 50 dans python grâce à la librairie pandas et sa fonction read_csv. On supprime par la suite la colonne des dates pour récupérer les rendements plus simplement.

```
import pandas as pd
import numpy as np
from sklearn import tree, preprocessing, metrics
from sklearn.ensemble import GradientBoostingClassifier, RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix

data = pd.read_csv("C:/Users/franc/Documents/ESILV/A5/Trading_Platform/Eurostoxx50_EOD_Clean.csv", sep=";", decimal
del data['Dates']
data = data[0:133]
```

Centrer et normer les séries des rendements

On calcule d'abord les rendements des séries financières définis par le ratio pour chaque jour j sur chaque colonne ou composante :

Rendement(j) = Différence de prix entre (j) et (j + 1) / Prix (j)

```
for col in data.columns:
    for i in range(len(data[col]) - 1):
        returns[col][i] = (float(data[col][i+1]) - float(data[col][i])) / float(data[col][i])
```

Par la suite on doit normaliser et centrer la matrice des rendements obtenus, c'est-à-dire calculer pour chaque composante de l'Eurostoxx 50 la moyenne des rendements et son écart-type puis appliquer à chaque ligne la nouvelle valeur :

Rendement(j) = (Rendement(j) – Moyenne(composant)) / Ecart-type(composant)

Pour ce faire, on utilise le package preprocessing et la fonction scale() :

```
#center and normalize returns
normalizedReturns = preprocessing.scale(returns)
normalizedReturns = np.delete(normalizedReturns, 47, 1)
```

Générer un arbre de décision pour prédire variations entre j et j+1 à partir des variations entre j et j-1

Pour générer un arbre de décision, il nous faut un vecteur target contenant les variables explicatives pour chaque ligne. Le but de l'arbre est de prédire la variation de prix, la variable explicative est alors égale à 0 ou 1 en fonction de la variation de prix observé entre j et j-1.

Pour construire ce vecteur target, on calcule la moyenne des rendements centrés normés de l'ensemble des composantes chaque jour. Si la moyenne est positive, c'est que de manière générale l'Eurostoxx 50 a subi une hausse des prix, on inscrit donc '1' comme variable explicative du vecteur target. Au contraire, si la moyenne est négative on inscrit '0' car les prix ont en moyenne baissé par rapport à la veille

```
#compute target vector
for i in range(len(normalizedReturns)):
    if normalizedReturns[i, np.arange(47)].mean() > 0:
        target.append(1)
    elif normalizedReturns[i, np.arange(47)].mean() < 0:
        target.append(0)
```

On peut désormais construire le modèle grâce à la librairie sklearn et au module de classification DecisionTreeClassifier(). Ce classifier prend en paramètre une matrice de données pour s'entrainer X et un vecteur target Y permettant de classer les élements en '1' ou '0'.

La fonction train_test_split() permet de diviser aléatoirement notre matrice de rendements normés en deux sous matrices avec la moitié de lignes ou jours ; une de test X_test et une de train X_train. On effectue la même chose pour le vecteur target et on récupère un y_train et un y_test.

```
X_train, X_test, y_train, y_test = train_test_split(normalizedReturns, target, test_size=0.5, random_state=42)
```

On passe donc X_train et y_train en paramètres au classifier, puis on peut évaluer la précision du modèle en comparant le vecteur y qui résulte du modèle soit y_predict avec le vrai vecteur target de x-test, soit y_test.

Ici, la précision du modèle trouvé est de 81,81%, ce qui veut dire que le vecteur de variables explicatives prédit par notre modèle possède 81% de mêmes valeurs que y_test.

```
clf = tree.DecisionTreeClassifier()
clf = clf.fit(X_train, y_train)
y_predict = clf.predict(X_test)
print("Accuracy:",metrics.accuracy_score(y_test, y_predict))
```

```
Accuracy: 0.8181818181818182
```

Améliorer la prédiction grâce au boosting

Pour le boosting, on divise la matrice de rendements et le vecteur target en deux sets de train et test. On utilise par la suite le module GradientBosstingClassifier() et on fit le modèle sur le X_train et le y_train.

Ce modèle crée une série d'arbres de décisions en se servant des erreurs des précédents arbres pour améliorer les suivants, il permet donc une meilleure précision qu'un arbre de décision basique et pour les échantillons de faible taille.

La précision du modèle dépend des paramètres inscrits lors de son instanciation comme :

- n_estimators, ou le nombre de phase de boosting à performer par défaut 100,
- learning_rate, soit le facteur qui réduit la contribution de chaque arbre,
- max_depth, qui définit le nombre maximal de nœuds dans chaque arbre.

```
#boosting
X_train, X_test, y_train, y_test = train_test_split(normalizedReturns, target, test_size=0.5

boosting = GradientBoostingClassifier(learning_rate = 0.2, n_estimators = 20, max_depth = 1)
boosting = boosting.fit(X_train, y_train)
y_predict = boosting.predict(X_test)

print('Accuracy : %f' % (boosting.score(X_test, y_test)))
```

```
Accuracy : 0.833333
```

Après plusieurs essais, la plus haute précision du modèle atteinte est 83,4% est obtenue avec n_estimators = 20, learning_rate = 0.2 et max_depth = 1.

Améliorer la précision grâce au random forest

Pour le random forest, on utilise le module RandomForestClassifier() avec des nouveaux sets de données train/test.

Ce modèle crée tout comme le bagging un ensemble d'arbres de décisions mais avec des sets de données légèrement différentes, ce qui permet de contrer le sur-apprentissage car il permet de baisser la variance de chaque set.

```
#random forest
X_train, X_test, y_train, y_test = train_test_split(normalizedReturns, t

randomForest = RandomForestClassifier(n_estimators = 20, max_depth = 2)
randomForest = randomForest.fit(X_train, y_train)
y_predict = randomForest.predict(X_test)

print('Accuracy : %f' % (randomForest.score(X_test, y_test)))
```

```
Accuracy : 0.893939
```

Générer les features_importances

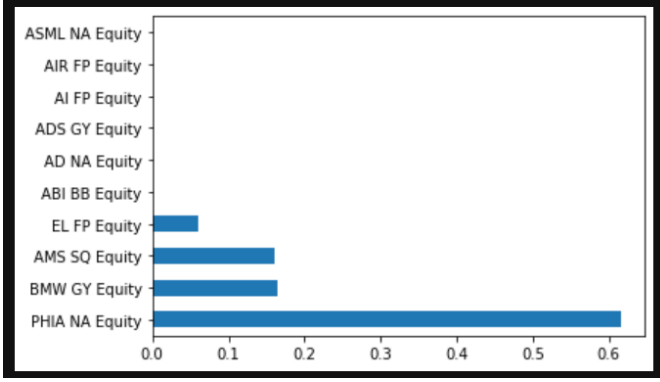
On cherche à visualiser l'importance de chacune des 50 composantes de l'Eurostoxx 50 dans la classification et donc la précision de la prédiction de chacun de ses modèles.

On crée alors une série pandas à partir de l'attribut feature_importances_ du modèle avec en index les noms de chacune des composantes. On plot par la suite les 10 composantes avec le plus gros poids dans la précision.

Par exemple, pour l'arbre de décision :

```
feat_importances = pd.Series(clf.feature_importances_, index=returns.iloc[:,0:47].columns)
feat_importances.nlargest(10).plot(kind='barh')
```

```
Accuracy: 0.8333333333333334
<matplotlib.axes._subplots.AxesSubplot at 0x2304b7dacf8>
```

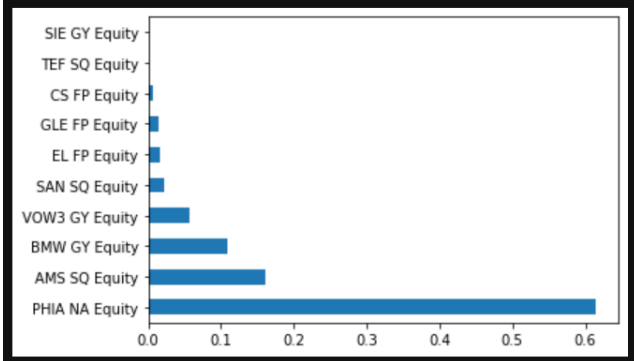


Ici, la précision dans les stocks est très asymétrique car c'est principalement le stock PHIA NA qui apporte 60% de précision au modèle.

Pour le boosting :

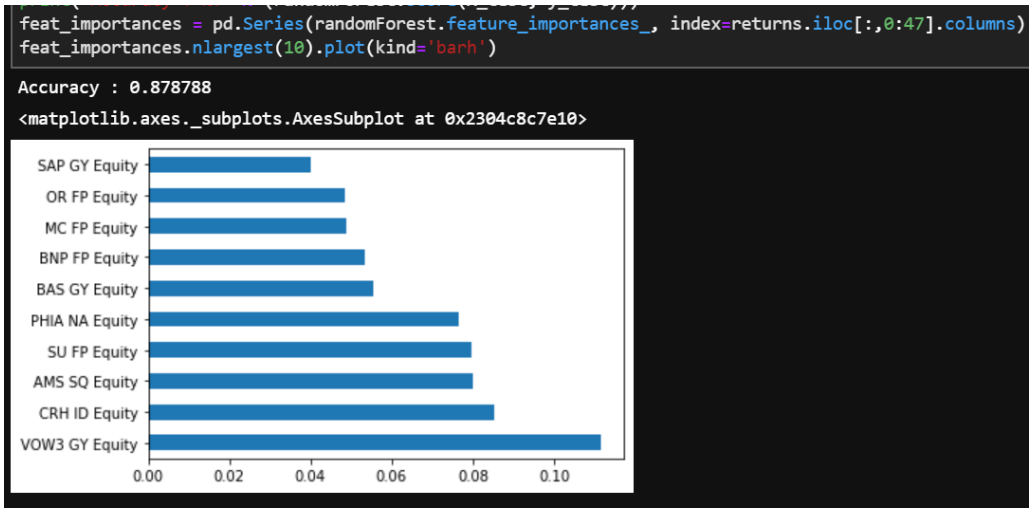
```
feat_importances = pd.Series(boosting.feature_importances_, index=returns.iloc[:,0:47].columns)
feat_importances.nlargest(10).plot(kind='barh')
```

```
Accuracy : 0.833333
<matplotlib.axes._subplots.AxesSubplot at 0x2304b87d2e8>
```



La précision est un peu plus distribuée mais c'est encore le stock PHIA NA qui apporte 60% de précision en moyenne.

Enfin, pour le random forest :

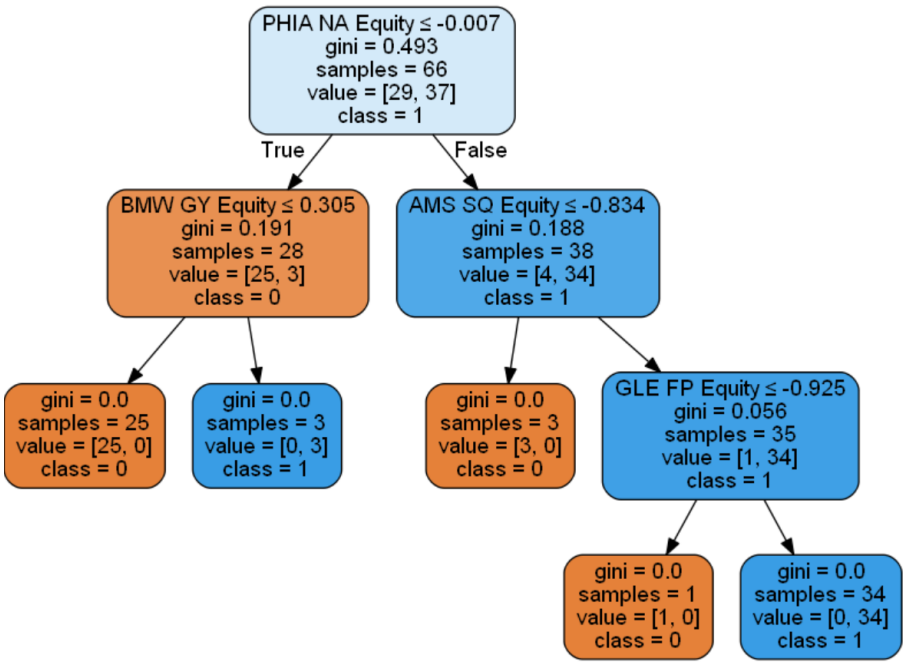


La précision est cette fois beaucoup plus distribuée et égale entre les dix plus gros stocks grâce aux sets de données redistribuées pour chaque arbre, ici c'est le stock VOW3 GY qui apporte le plus de précision au modèle, soit près de 12%.

En déduire un panier de stock longs qui surperforme l'Eurostoxx

Pour déduire un panier de stock longs only qui surperforme l'Eurostoxx 50, il nous faut visualiser notre arbre de décision grâce à la fonction export_graphviz().

Si on visualise le graph de l'arbre de décision :



Ont peut observer plusieurs paramètres sur chaque nœud :

- Le coefficient ‘gini’ désigne un critère d’impureté ou d’homogénéité du nœud, il augmente s’il y a beaucoup d’instances de plusieurs classes et est égal à 0 s’il n’y a que des instances de la même classe,
- Le vecteur ‘value’ a deux valeurs pour indiquer combien d’instances il y a de chaque classe ‘0’ et ‘1’ dans chaque nœud,
- La classe : ‘0’ pour une baisse des prix, et ’1’ pour une hausse,
- Le critère de classification du nœud, si l’échantillon satisfait la condition, il va dans le nœud fils de gauche, sinon celui de droite.

Sachant que nous cherchons à trouver un nœud avec 100% de valeurs de classe ‘1’, ou un nœud bleu de coefficient ‘gini’ nul avec le plus de samples, on peut observer le dernier nœud en bas à droite.

Ici, on obtient 34 jours sur les 66 passés en test appartenant à la classe ‘1’ et divisés après seulement trois critères :

- Nœud 1 : le rendement centré normé du stock ‘PHIA NA’ est supérieur à -0.007,
- Nœud 2 : le rendement centré normé du stock ‘AMS SQ’ est supérieur à -0.834,
- Nœud 3 : le rendement centré normé du stock ‘GLE FP’ est supérieur à -0.925.

Si ces trois stocks possèdent des rendements centrés supérieurs à ces niveaux, alors le modèle prédit une hausse des prix de l’Eurostoxx.