Machine Learning

Martin Lefevre
Thomas Perier
Ambroise Warnery

Suppression colonne NaN
Reconnaissance expression régulière: date

Parcourt toutes les colonnes, calcule la proportion de NaN, et supprime sur place celles dépassant le seuil de 20% afin de préparer les données pour l'analyse.

```
def clean_NaN(df):
    colonnes_suppr = []
    lignes_suppr = []
```

Détection automatique de champs de date dans l'ensemble de données. Identification flexible et robuste des différents formats de date.

```
#fonction date qu'on utilise apres dans la fonction qui detecte le type de variable
def is_date(value):
    # Liste de regex pour les formats de dates les plus connus
    date_regex_patterns = [
        r'\d{4}-\d{2}-\d{2}',
```

Suppression colonne NaN (2)

```
[13] #ici on commence par appliquer notre fonction de nettoyage
     clean_NaN(titanic)
```

La ou les colonnes 'Cabin' ont été supprimés

```
titanic.dtypes
```

```
Survived    category
Pclass      category
Sex         category
Age          float64
SibSp       category
Parch       category
Fare         float64
Embarked    category
dtype: object
```

Détermination type variables pour chaque colonnes

La fonction offre une méthode automatique pour la détection du type de données, couvrant les types les plus courants rencontrés dans l'analyse de données : catégorique, temporel, numérique (flottant et entier), et textuel.

Automatisation du processus de détermination des types de données dans les colonnes d'un DataFrame.

```
#fonction qui détecte automatiquement le type de variable d'une colonne
def get_column_variable_type(column):
  if column.nunique() < 11:
    return 'category'
  elif column.apply(is date).all() == True:
```

```
#fonction pour récupérer les types de variables de touts nos colonnes dans l'ordre
def data_type_columns(data: pd.DataFrame):
    type_var_colonnes = []
    for column in data.columns:
        type_var_colonnes.append(get_column_variable_type(data[column]))
    return type_var_colonnes
```

Test unitaire de la fonction `data_type_columns`

```python
# TEST UNITAIRE DE get_column_variable_type()
# on utilise un dataset écrit en dure créer spécifiquement pour tester toutes les sorties possibles de la fonction get_column_variable_type()
# on a aussi une listes avec les réponses attendues sur notre dataset de test
# finalement on compare les deux listes pour voir si elles sont égales
# si toutes les sorties sont corrects, ont considère que notre fonction est bonne, sinon, on raise une erreure, notre fonction bug

data = {
    'bool_column': [True, False, True, False, True, False, True, False, True, False, True],
    'category_column': ['A', 'B', 'A', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'],
    'date_column': pd.date_range('2023-01-01', periods=11),
    'str_column': ['apple', 'banana', 'orange', 'pear', 'grape', 'pineapple', 'lemon', 'melon', 'peach', 'plum', 'kiwi'],
    'float_column': [1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9, 10.1, 11.11],
    'int_column': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11],
    'else_column' : [2, 'B', 'zigzag', True, 3, 4, 5, 6, 7, 8, 9]
}

data_test_unitaire = pd.DataFrame(data)
data_type_attendues = ['category', 'category', 'datetime', 'str', 'float', 'int', 'str']

def test_unitaire(data_test_unitaire, data_type_attendues):
    if data_type_columns(data_test_unitaire) != data_type_attendues:
        raise ValueError("Il y a un problème, la fonction data_type_columns() ne retourne pas les bons résultats")
    else:
        print("Test unitaire validé")
```

```
[9] test_unitaire(data_test_unitaire, data_type_attendues)

    Test unitaire validé
```

La fonction zip

La fonction “Zip” en Python est utilisée pour agglomérer deux ou plusieurs séquences (comme des listes, des tuples ou des ensembles de colonnes d'un DataFrame) élément par élément, créant un nouvel itérable de tuples. Chaque tuple contient un élément de chaque séquence, appariés ensemble selon leur position dans chaque séquence.

Il est important de noter que si les séquences passées “Zip” ne sont pas de la même longueur, alors on s'arrêtera à la fin de la plus courte séquence, ignorant ainsi les éléments supplémentaires dans les autres séquences. Cela signifie que si “data_types” contient moins d'éléments que “df_columns”, alors certaines colonnes ne seront pas affectées par la fonction “change_dtypes”.

```python
#fonction pour appliquer les bonnes catégories à nos variables
def change_dtypes(df, data_types):
    for column, dtype in zip(df.columns, data_types):
        df[column] = df[column].astype(dtype)
```

Le type de variable “Object”

A la fin de notre classification, on a constaté qu’ils nous restaient des variables de type “Object”. On va donc les supprimer pour ne pas être gêné pour la suite du projet. La fonction modifie donc le DataFrame original en supprimant toutes les colonnes dont le type est 'object'. Cette opération est courante dans le prétraitement des données pour les modèles de machine learning, où les colonnes de type 'object' peuvent correspondre à des données textuelles ou catégorielles qui nécessitent une transformation avant d'être utilisées dans un modèle. Voici donc la fonction réalisée :

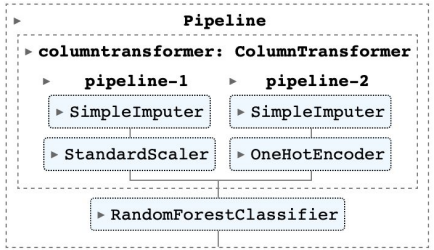
```
#fonction pour supprimer les colonnes de type 'object', c'est celle qu'on a pas réussi à classifier
def remove_object_columns(df):
    object_columns = df.select_dtypes(include=['object']).columns
    df.drop(columns=object_columns, inplace=True)
```

Création de la pipeline de prétraitement

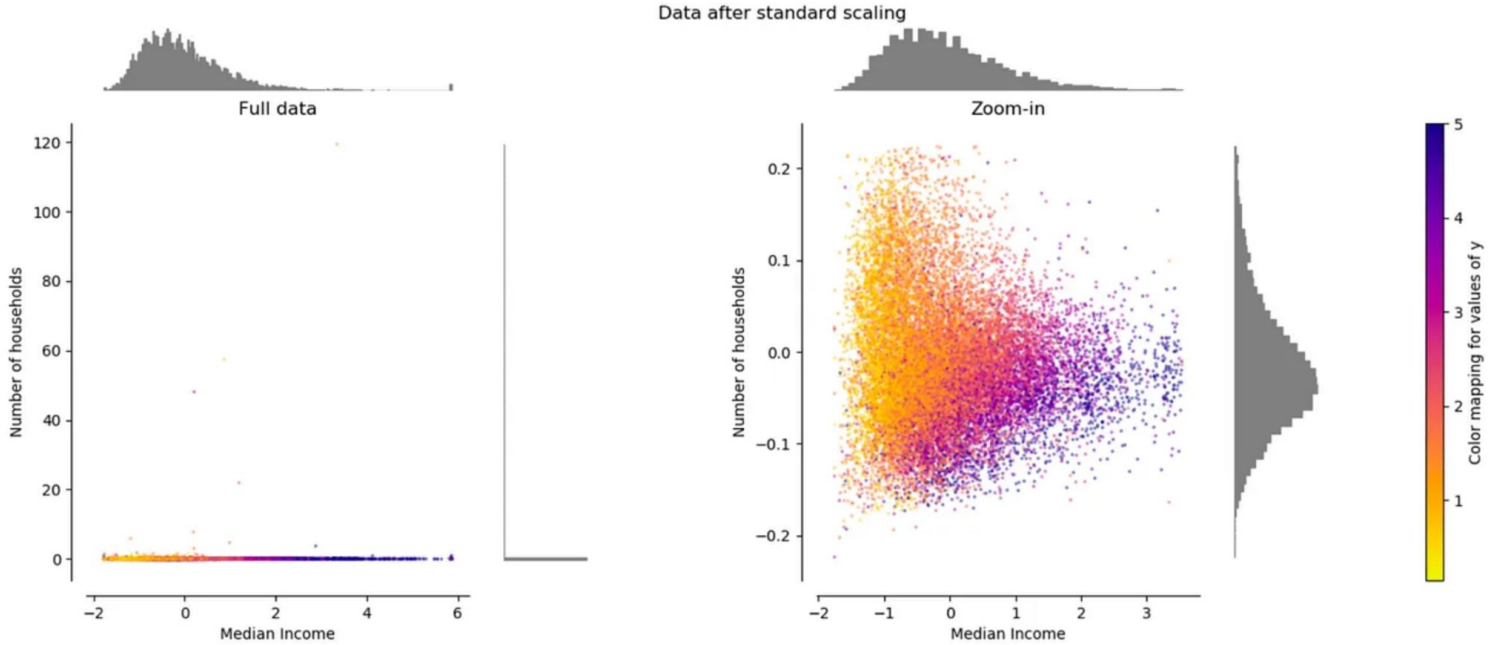
```
[12] # creer la pipeline de preprocessing
     def pre_process_pipeline():
       numerical_features = make_column_selector(dtype_include=np.number)
       categorical_features = make_column_selector(dtype_include= 'category')
       bool_features = make_column_selector(dtype_include= 'bool')

       numerical_pipeline = make_pipeline(SimpleImputer(strategy='mean'),
                                          StandardScaler())
       categorical_pipeline = make_pipeline(SimpleImputer(strategy='most_frequent'),
                                          OneHotEncoder())

       preprocessor = make_column_transformer((numerical_pipeline, numerical_feature
                                       (categorical_pipeline, categorical_features)
       #model = make_pipeline(preprocessor, LazyClassifier(verbose=0)))

       #df_preprocessed = preprocessor.fit_transform()

       return preprocessor
```

```
[53] model_RF_titanic = make_pipeline(titanic_preprocessor, RandomForest
```

```
model_RF_titanic.fit(X_train, y_train)
```



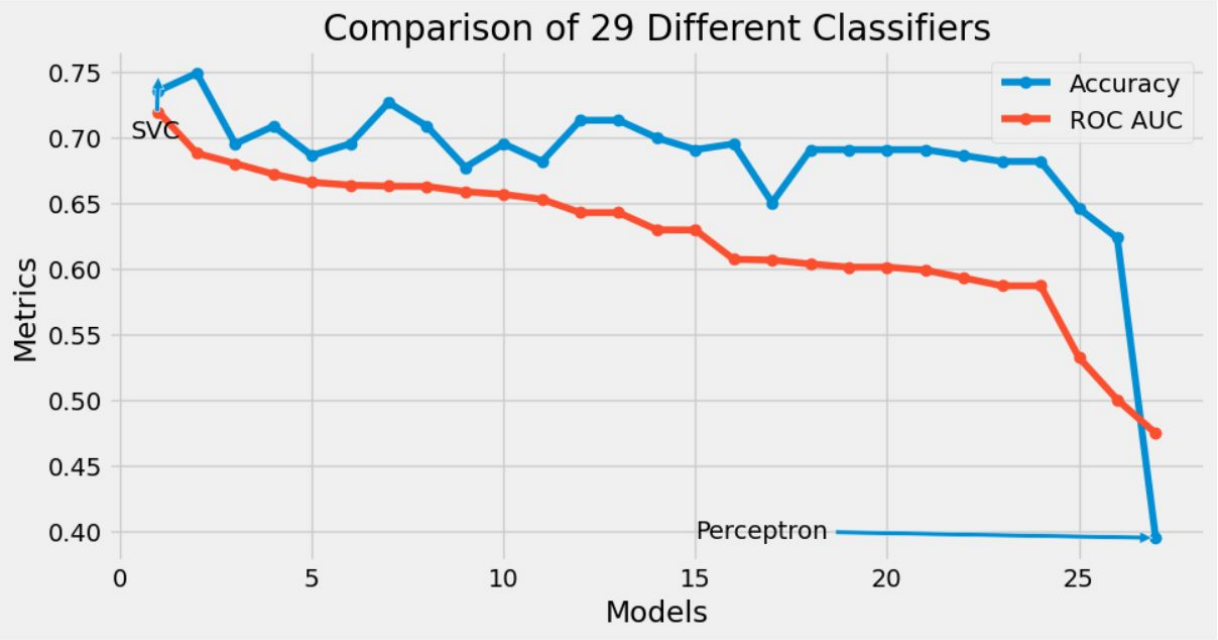
StandardScaler()



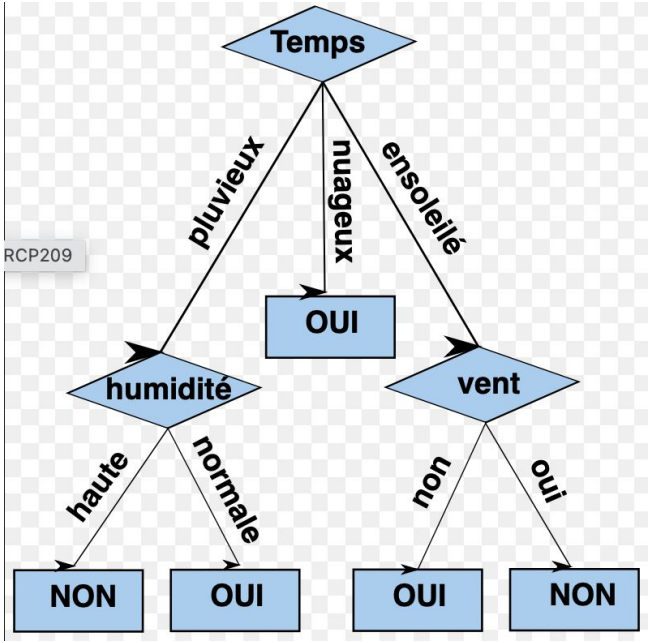
La bibliothèque “Lazypredict”

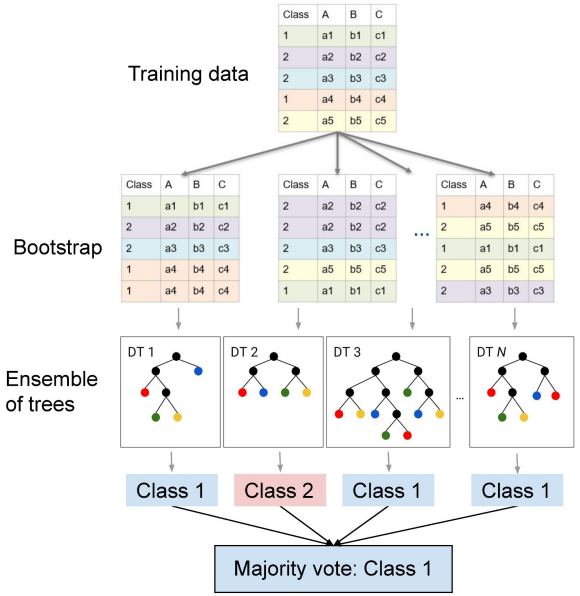
- Elle permet de tester rapidement plusieurs modèles de machine learning
- Pour l’utiliser, création d’une base de test et d'entraînement sur les données brutes donc pas encore normalisé.
- Obtention des modèles de machine learning les plus performants selon l’accuracy, la “balanced accuracy”, la “ROC AUC”, le “F1 score”, et le “TIme taken”.
- On constate que c’est le modèle “Random Forest” le plus adapté (0,74 -> accuracy et 0,72 -> ROC AUC)

Comparaison de nos 29 modèles



RandomForest





Optimisation RF

- Nombre d'arbre
- Profondeur max des arbres

mais aussi

- Nombre min d'individu sur une feuille
- etc.

```
pipe_params = {
    'randomforestclassifier__n_estimators': [100, 200, 300],
    'randomforestclassifier__max_depth': [None, 10, 20, 30],
}



gridsearch = GridSearchCV(
    model_RF_titanic, param_grid=pipe_params, cv=3, n_jobs=-1, verbose=1000
)

gridsearch.fit(X_train, y_train)
```

Optimisation RF

```
[119] best_hyperparameters = gridsearch.best_params_
      print("Best hyperparameters:", best_hyperparameters)
      best_score = gridsearch.best_score_
      print("Best cross-validation score:", best_score)
```

```
Best hyperparameters: {'randomforestclassifier__max_depth': None, 'randomforestclassifier__n_estimators': 100}
Best cross-validation score: nan
```

```
[121] score_gridsearch = gridsearch.score(X_test, y_test)
      score_gridsearch
```

```
0.8251121076233184
```

Ecart type et évaluation de la difficulté

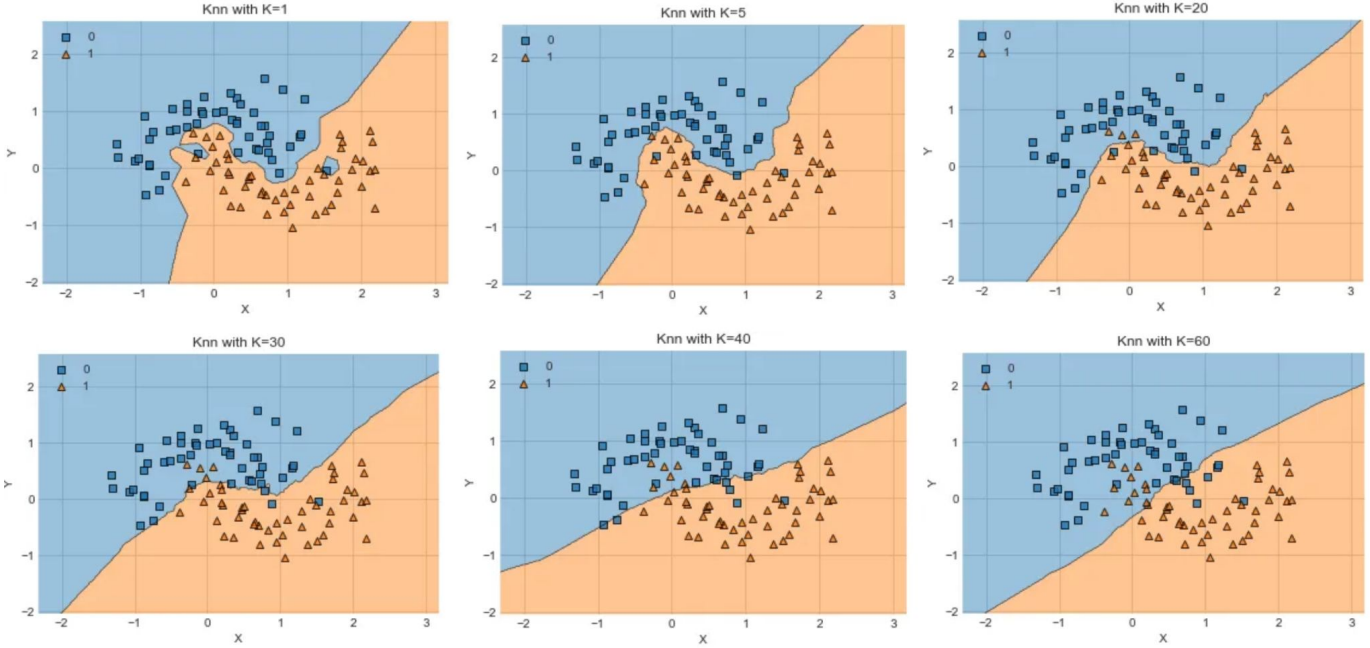
```python
#Ici on calcule un intervalle de confiance de la précision de notre modele
X_test = pd.DataFrame(X_test)

# Reset the index of X_test to ensure it starts from 0
X_test.reset_index(drop=True, inplace=True)

# Define the number of bootstrap iterations
n_bootstraps = 1000

# Initialize an array to store accuracy scores from each bootstrap iteration
bootstrap_scores = np.zeros(n_bootstraps)

# Perform bootstrapping
for i in range(n_bootstraps):
    # Generate a bootstrap sample (with replacement) from the test set
    bootstrap_indices = np.random.choice(len(X_test), size=len(X_test), replace=True
    X_bootstrap = X_test.iloc[bootstrap_indices]
    y_bootstrap = y_test.iloc[bootstrap_indices]

    # Evaluate the model on the bootstrap sample and compute accuracy
    score = gridsearch.score(X_bootstrap, y_bootstrap)

    # Store the accuracy score
    bootstrap_scores[i] = score

# Calculate the confidence interval
confidence_interval = np.percentile(bootstrap_scores, [2.5, 97.5])

# Print the confidence interval
print("95% Confidence Interval for Accuracy: [{:.4f}, {:.4f}]".format(confidence_interval[0], confidence_interval[1]))
```

```
95% Confidence Interval for Accuracy: [0.7758, 0.8789]
```

```python
#Apres avoir calculé l'intervalle de confiance, si celui ci est élevé (>10%), on affiche que le probleme est complexe,
# sinon qu'il est facile
interval_diff = confidence_interval[1] - confidence_interval[0]

seuil = 0.1 * (confidence_interval[1] + confidence_interval[0]) / 2

if interval_diff < seuil:
    print("Le probleme est facile")
else:
    print("Le probleme est dur")
```

```
Le probleme est dur
```

KNN


Knn with K=1
Knn with K=5
Knn with K=20
Knn with K=30
Knn with K=40
Knn with K=60
0
1
X
Y

Optimisation KNN

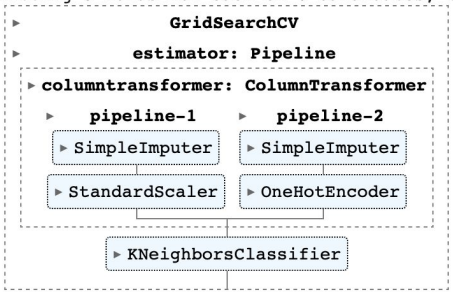
- Nombre de voisin
- Pondération du vote par la distance

mais aussi

- Mesure de distance utilisée (Manhattan, Euclidienne,...)
- etc.

```
pipe_params = {
    'kneighborsclassifier__n_neighbors': [3, 5, 7],
    'kneighborsclassifier__weights': ['uniform', 'distance']
}



gridsearch = GridSearchCV(                    (parameter) cv: Any | None
    model_KNN_brain, param_grid=pipe_params, cv=3, n_jobs=-1, verbose=1000
)

gridsearch.fit(X_train, y_train)
```

```
Fitting 3 folds for each of 6 candidates, totalling 18 fits
```



Optimisation KNN

```
best_hyperparameters = gridsearch.best_params_
print("Best hyperparameters:", best_hyperparameters)
best_score = gridsearch.best_score_
print("Best cross-validation score:", best_score)
```

```
Best hyperparameters: {'kneighborsclassifier__n_neighbors': 3, 'kneighborsclassifier__weights': 'uniform'}
Best cross-validation score: nan
```

```
score_KNN = gridsearch.score(X_test, y_test)
score_KNN
```

```
0.9428794992175273
```

Ecart type et évaluation de la difficulté

```python
#Ici on calcule un intervalle de confiance de la précision de notre modele
X_test = pd.DataFrame(X_test)

# Reset the index of X_test to ensure it starts from 0
X_test.reset_index(drop=True, inplace=True)

# Define the number of bootstrap iterations
n_bootstraps = 1000

# Initialize an array to store accuracy scores from each bootstrap iteration
bootstrap_scores = np.zeros(n_bootstraps)

# Perform bootstrapping
for i in range(n_bootstraps):
    # Generate a bootstrap sample (with replacement) from the test set
    bootstrap_indices = np.random.choice(len(X_test), size=len(X_test), replac
    X_bootstrap = X_test.iloc[bootstrap_indices]
    y_bootstrap = y_test.iloc[bootstrap_indices]

    # Evaluate the model on the bootstrap sample and compute accuracy
    score = gridsearch.score(X_bootstrap, y_bootstrap)

    # Store the accuracy score
    bootstrap_scores[i] = score

# Calculate the confidence interval
confidence_interval = np.percentile(bootstrap_scores, [2.5, 97.5])

# Print the confidence interval
print("95% Confidence Interval for Accuracy: [{:.4f}, {:.4f}]".format(confidence_interval[0], confidence_interval[1]))
```

```
95% Confidence Interval for Accuracy: [0.9296, 0.9546]
```

```python
#Apres avoir calculé l'intervalle de confiance, si celui ci est élevé (>10%), on affiche que le probleme est complexe,
# sinon qu'il est facile
interval_diff = confidence_interval[1] - confidence_interval[0]

seuil = 0.1 * (confidence_interval[1] + confidence_interval[0]) / 2

if interval_diff < seuil:
    print("Le probleme est facile")
else:
    print("Le probleme est dur")
```

```
Le probleme est facile
```

Merci de votre écoute