Problème de prédiction multi-classes sur des données de jeux en ligne

Table des matières

1 Partie A

1.1 Méthodologie

1.1.1 Méthodes testées

Tout d'abord, quelques méthodes différentes d'apprentissage automatique ont été testées afin de déterminer le modèle optimal. Pour ce faire, quelques packages d'autoML ont été utilisés avant de commencer les recherches approfondies. Par exemple, FLAML est un package Python très performant pour les modèles de type Gradient Boosting, ce qui a permi d'obtenir un premier modèle via le package CatBoost. Une forêt aléatoire a aussi été ajustée, ainsi qu'un réseau de neurones. Dans tous les cas, certains hyper-paramètres ont été optimisés. Les meilleures performances obtenues pour chaque modèle sont présentées dans le Tableau 1 ci-dessous. On constate que la méthode qui a eu le plus de succès est le réseau de neurones et elle a donc été retenue. Les détails de la méthodologie employée sont précisés dans la section suivante.

Modèle	Précision (%)
Forêt aléatoire	75.1
Gradient Boosting	80.9
Gradient Boosting avec one-vs-one	82.3
Réseau de neurones	82.4

Table 1: Meilleure performance pour chaque méthode testée en termes de précision.

1.1.2 Réseau de neurones

La première étape consistait à transformer les variables catégorielles en variables indicatrices à l'aide d'un encodage one-hot, effectué grâce à la fonction *get_dummies*. Les données d'entraînement ont ensuite été divisées aléatoirement en deux ensembles : ensemble d'entraînement (4000 observations) et ensemble de validation (1000 observations). La variable cible a également été transformée en indicatrices via la fonction *to_categorical* du module Keras. Les données ont été standardisées pour avoir une moyenne de 0 et un écart-type de 1.

La prochaine étape consistait à optimiser les hyper-paramètres grâce à une recherche par grille (*GridSearchCV* de Keras). Les hyper-paramètres optimisés incluent : le nombre de couches cachées, le nombre de neurones par couche, la présence ou non de dropout sur les couches cachées, ainsi que le taux d'apprentissage. Une validation croisée à 3 plis a été réalisée pour chaque combinaison d'hyper-paramètres, en maximisant la précision moyenne. Un mécanisme d'arrêt anticipé (*EarlyStopping*) a été utilisé pour arrêter l'entraînement si la perte sur des nouvelles données ne diminuait pas pendant 5 époques consécutives. Il faut noter que 10% de l'ensemble d'entraînement a été mis de côté

pour surveiller cette performance. Cela permettait d'éviter le surapprentissage qui aurait été possible en laissant le nombre d'époques fixe.

Afin de creuser davantage, une brève évaluation des variables après la première optimisation a été faite. En faisant les matrices de corrélation et les VIF (*Variance Inflation Factors*, la variable *numelements* semblait être corrélée à *skill1* et qui avait un VIF de 8.62, ce qui aurait pu indiquer diverses pertes dans la qualité de notre modèle, notamment à cause de la redondance de ce que les variables expriment comme signal prédictif. Pour autant, la précision de notre modèle n'augmentait pas en la retirant; elle a donc été gardée puisque notre objectif était de trouver la meilleure précision.

Une fois les hyper-paramètres optimaux identifiés, le modèle a été entraîné à nouveau sur l'ensemble d'entraînement pour 500 époques. Les courbes de perte et de précision ont été tracées pour suivre les performances sur l'ensemble de validation à chaque époque. Finalement, le modèle a été réentraîné sur toutes les données d'entraînement disponibles en utilisant le nombre d'époques qui maximisait la précision sur l'ensemble de validation.

Le modèle final retenu pour les prédictions est un réseau de neurones comportant : 3 couches cachées de 256 neurones avec du dropout, des fonctions d'activation ReLU et de la régularisation ridge. L'optimiseur utilisé est *Adam* avec un taux d'apprentissage de 0,001. Une fonction d'activation softmax a été appliquée à la couche de sortie pour obtenir des probabilités pour chaque classe sommant à un total de 1. Pour l'entraînement, la taille de batch considérée est 512 et le nombre d'époques choisi est 278.

1.1.3 Code Python (réseau de neurones)

Cette section présente le code utilisé pour la mise en oeuvre du réseau de neurones. Toutefois, certaines parties ou lignes jugées non pertinentes sont omises, le code n'est donc pas affiché dans son intégralité.

```
import numpy as np
import pandas as pd
import random
import tensorflow as tf

from tensorflow import keras
from tensorflow.keras import regularizers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Dense, Dropout
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
from scikeras.wrappers import KerasClassifier
from keras.callbacks import EarlyStopping

# Pour la reproductibilité
np.random.seed(7)
random.seed(7)
tf.random.set_seed(7)

# Création des indicatrices pour les variables catégorielles
namescat = ["country", "difflevel"]
datrain_x = datrain.drop(columns=["y"])
datest_x = datest.drop(columns=["id"])
n_train = datrain_x.shape[0]
datall_x = pd.concat([datrain_x, datest_x], axis=0)
datall_x_dum = pd.get_dummies(datall_x, columns=namescat,
                              dtype='int', drop_first=False)
datrain_x_dum = datall_x_dum.iloc[:n_train, :]
datest_x_dum = datall_x_dum.iloc[n_train:, :]

# Préparation des données d'entraînement et de validation
rand = np.random.choice(5000, size=5000, replace=False)
x_train = datrain_x_dum.iloc[rand[:4000], :].values
y_train = datrain.iloc[rand[:4000], :]["y"].values - 1  # Classes 0-3
x_valid = datrain_x_dum.iloc[rand[4000:], :].values
y_valid = datrain.iloc[rand[4000:], :]["y"].values - 1  # Classes 0-3
```

```
# Transformation de la variable cible avec one-hot encoding
y_train = keras.utils.to_categorical(y_train)
y_valid = keras.utils.to_categorical(y_valid)

# Standardisation des données
scaler = StandardScaler()
x_train = scaler.fit_transform(x_train)
x_valid = scaler.transform(x_valid)
```

```
# Fonction pour construire le modèle
def build_model(num_layers=2, num_neurons=32, dropout_rate=0,
                learning_rate=0.001):

    model = Sequential()
    model.add(Input(shape=(23,)))
    # Ajouter les couches une par une selon le nombre
    for i in range(num_layers):
        model.add(Dense(num_neurons, activation="relu",
                        kernel_regularizer=regularizers.l2(0.002)))
        model.add(Dropout(dropout_rate))

    model.add(Dense(4, activation="softmax"))

    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),
        loss="categorical_crossentropy",
        metrics=["accuracy"])

    return model

# Mécanisme d'arrêt anticipé
early_stopping = EarlyStopping(monitor='val_loss', patience=5,
                               restore_best_weights=True, verbose=0)

# Créer un KerasClassifier pour l'utiliser avec GridSearchCV
keras_clf = KerasClassifier(build_fn=build_model, epochs=500, batch_size=512,
                            callbacks=[early_stopping], verbose=0)

# Définir la grille de recherche d'hyperparamètres
param_grid = {"model__num_layers": [2, 3],
              "model__num_neurons": [64, 128, 256, 512, 1024],
              "model__dropout_rate": [0, 0.5],
              "model__learning_rate": [0.0005, 0.001]}

# Configuration de GridSearchCV
grid = GridSearchCV(estimator=keras_clf, param_grid=param_grid,
                    cv=3, # Validation croisée à 3 plis
                    scoring='accuracy', verbose=0)

# Lancement de la recherche des meilleurs hyperparamètres
grid_result = grid.fit(x_train, y_train, validation_split=0.1)
best_params = grid_result.best_params_

# Afficher les meilleurs paramètres et le score correspondant
print(f"Meilleurs paramètres: {best_params}")
print(f"Meilleure précision avec validation croisée: {grid_result.best_score_}")
```

```
Meilleurs paramètres: {'model__dropout_rate': 0.5, 'model__learning_rate':
0.001, 'model__num_layers': 3, 'model__num_neurons': 256}
Meilleure précision avec validation croisée: 0.7752483473192137
```

```
# Construire un nouveau modèle avec les meilleurs hyperparamètres
final_model = build_model(
    num_layers=best_params['model__num_layers'],
    num_neurons=best_params['model__num_neurons'],
    dropout_rate=best_params['model__dropout_rate'],
    learning_rate=best_params['model__learning_rate'])

# Entraîner le modèle pour vérifier le nombre d'époques optimal
histrn = final_model.fit(x_train, y_train, epochs = 500, batch_size = 512,
                         validation_data = (x_valid, y_valid), verbose = 0)

# Extraction de l'historique d'entraînement et identification de l'époque optimal
history = histrn.history
best_epoch = np.argmax(history['val_accuracy']) + 1
print(f"Meilleure précision sur l'ensemble de validation atteinte à l'époque :␣
 ↪{best_epoch}")
```

```
Meilleure précision sur l'ensemble de validation atteinte à l'époque : 278
```

```
# Préparation des données finales
x_train = datrain_x_dum
y_train = datrain["y"] - 1  # Classes 0-3
x_test = datest_x_dum
y_train = keras.utils.to_categorical(y_train)

# Standardisation des données
scaler = StandardScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)

# Réentraîner sur l'ensemble complet des données d'entraînement et faire les␣
 ↪prédictions
final_model.fit(x_train, y_train, batch_size=512, epochs=best_epoch, verbose=0)
y_pred = final_model.predict(x_test)
final_predictions = np.argmax(y_pred, axis=1) + 1
```

1.1.4 Gradient Boosting avec approche one-vs-one

Le deuxième modèle présenté a été exploré avec le même niveau de détail que le réseau de neurones ci-dessus. Afin d'alléger le texte, seules les différences seront mentionnées.

Le principe pour implémenter une approche one-vs-one est de séparer chaque catégorie de la variable réponse et d'y appliquer un modèle au choix (ici un Gradient Boosting modélisé par *CatBoost*) afin de les comparer entre elles. Le package de scikit-learn contient une fonction préconçue permettant d'appliquer facilement une approche one-vs-one. Son script a été vérifié pour s'assurer qu'il compare chaque paire de catégories de la variable Y avec le modèle spécifié. Pour le critère de décision, la fonction effectue une transformation monotone et sélectionne la catégorie ayant la probabilité la plus élevée, ce qui convient à cette analyse. Il est important de mentionner également que CatBoost requiert certaines étapes de préparation des données, à savoir que les prédicteurs catégoriels soient spécifiées ainsi que d'encoder les catégories de la variable réponse en commençant par 0 (et non 1 dans notre jeu de données).

1.1.5 Code Python (Gradient Boosting)

```python
# Preparation spécifique à catboost
categorical_columns = ['difflevel', 'country']
for col in categorical_columns:
    X_train[col] = X_train[col].astype('category')
    X_val[col] = X_val[col].astype('category')
    test_data[col] = test_data[col].astype('category')

# Initier la méthode one-vs-one
ovo = OneVsOneClassifier(catboost_model)
# Ajuster le modèle
ovo.fit(X_train, y_train)
# Faire des prédictions sur les données de validation
yhat = ovo.predict(X_val)

yhat_accuracy = accuracy_score(y_val, yhat)
print(f'Validation Accuracy: {yhat_accuracy}')
```

2 Partie B

2.1 Présentation des ressources

Une simple recherche Web à propos de l'optimisation des hyper-paramètres ou « hyper-parameter optimization » (HPO) permet d'accéder à une multitude d'informations sur le sujet. Il est néanmoins important de bien filtrer ces sources pour assurer la qualité de l'explication et l'analyse. Une des sources principales pour cette introduction au HPO est la revue de Tong Yu & Hong Zhu publiée sur arXiv, une plateforme communautaire supportée principalement par l'université Cornell, la Fondation Simons, des institutions membres et des donateurs.

Cette revue présente plusieurs méthodes d'optimisation d'hyper-paramètres, mais les 3 méthodes qui seront détaillées sont l'échantillonnage par grille, l'échantillonnage aléatoire et l'optimisation bayésienne. Pour valider les informations présentées, la publication de Li Yang et Abdallah Shami dans le volume 415 du journal Neurocomputing a été utilisée. Enfin, un article traitant des méthodes d'optimisation des hyper-paramètres prises en charge par l'outil Azure Machine Learning a guidé la sélection de ces approches. Il est à noter que l'optimisation bayésienne est basée sur l'échantillonnage bayésien, celui-ci étant supporté par l'outil de Microsoft. Cela permet de mettre dans une perspective plus pratique que théorique les méthodes expliquées dans les autres sources.

2.2 Méthodes d'optimisation des hyper-paramètres

L'échantillonnage par grille est une méthode de HPO qui effectue une exploration systématique des valeurs d'hyper-paramètres contenues dans l'espace (la grille) défini par l'utilisateur. Il évalue dans le modèle qui sera utilisé toutes les combinaisons possibles à l'aide du produit cartésien pour arriver à un résultat qui est optimal dans l'espace délimité. Cette méthode est très répandue vue sa simplicité

mathématique et puisqu'il est possible de paralléliser. Cependant, elle implique l'évaluation de toutes les possibilités, ce qui la rend très demandante au niveau computationnel. De plus, cette méthode souffre du *curse of dimensionality* puisque chaque ajout d'une dimension (ajout d'un hyper-paramètre) augmente de façon exponentielle le nombre de calculs à faire. Elle n'est donc recommandée que lorsque les ressources de calculs ne sont pas un enjeu, que l'espace défini pour chaque hyper-paramètre est relativement faible ou connu, ou qu'il y a peu d'hyper-paramètres à optimiser.

L'échantillonnage aléatoire contient les mêmes étapes que l'échantillonnage par grille, mais, au lieu de définir les valeurs exactes à évaluer, l'utilisateur définit des intervalles ou des distributions à partir desquelles des valeurs aléatoires sont sélectionnées. Ces valeurs aléatoires sont ensuite utilisées pour former les combinaisons à tester. Cette méthode est donc moins exigeante au niveau computationnel que par grille, mais elle demeure quand même relativement coûteuse. Elle conserve un des avantages de l'échantillonnage par grille en étant parallélisable et peut servir de préfiltre de l'espace possible pour les hyper-paramètres avant d'employer une méthode plus exhaustive comme l'échantillonnage par grille.

Finalement, l'optimisation bayésienne est une méthode séquentielle basée sur un modèle substitut pour évaluer la fonction objective. Cette dernière doit minimalement prendre les hyper-paramètres comme intrants et produire une métrique de performance comme extrant. Le modèle substitut peut être construit de plusieurs façons, souvent à l'aide d'une forêt aléatoire, un *Gaussian Process (GP)* ou un *Tree-structured Parzen Estimator (TPE)*. Ces éléments sont expliqués en détail dans la publication de Li Yang et Abdallah Shami. Par la suite, il faut définir une fonction d'acquisition qui évalue les différents points sur le modèle substitut. Enfin, ces points sont utilisés pour mettre à jour le modèle substitut et répéter ces opérations jusqu'à l'atteinte d'un point optimal. Cette méthode équilibre l'exploration et l'exploitation et est particulièrement efficace pour optimiser des fonctions complexes.

En résumé, ces trois méthodes permettent d'optimiser les hyper-paramètres avec des méthodes différentes. L'échantillonnage par grille est simple et facile à implémenter, mais peut demander un temps de calcul excessivement long. L'échantillonnage aléatoire est une approche plus flexible et légèrement moins coûteuse. L'optimisation bayésienne est une technique plus poussée, mais est l'option à favoriser pour des fonctions complexes. D'autres approches décrites dans nos sources telles que le « Gradient-based optimization » ou les « Multi-fidelity optimization algorithms » peuvent être adaptées dans certaines situations.

2.3 Code

L'optimisation bayésienne des hyper-paramètres d'une forêt aléatoire a été appliquée avec les données de l'exemple de jeux en ligne du cours comportant un Y continu. La performance obtenue pour la forêt aléatoire avec les hyper-paramètres optimisés est nettement meilleure (voir Tableau 2 ci-dessous).

Modèle	MSE	R^2	MAE	MAPE
Forêt aléatoire de base	39.46198	0.7402770	4.788810	17.47923
Forêt aléatoire optimisée	32.13497	0.7885005	4.356472	15.7542

Table 2: Comparaison des performances des modèles

```r
library(ranger)
library(rBayesianOptimization) set.seed(7)

# Importation des données
datrain <- read.table("clv_mobile_data_train.txt",header=TRUE) datest <-
read.table("clv_mobile_data_test.txt",header=TRUE)

# Transformation des variables catégorielles en facteur datrain$difflevel <-
factor(datrain$difflevel) datrain$country <- factor(datrain$country)
datest$difflevel <- factor(datest$difflevel) datest$country <-
factor(datest$country)

# Entraîner la forêt aléatoire de base
rf_model <- ranger(y~.,data=datrain, importance="permutation", respect.unordered.factors="partition")

# Définir la fonction objectif pour l'optimisation
rf_optimization <- function(mtry, min_node_size, sample_fraction) {

  # Entraîner le modèle
  rf_model <- ranger(y ~ ., data = datrain, mtry = floor(mtry), # Convertir mtry en entier
                     min.node.size = floor(min_node_size), # Convertir min.node.size en entier sample.fraction =
                     sample_fraction, # Fraction d'échantillonnage importance = "permutation",
                     respect.unordered.factors = "partition", num.trees = 500,
                     oob.error = TRUE) # Activer le calcul de l'erreur OOB

  # Calculer et retourné le R2 basé sur l'OOB
  ss_total <- sum((datrain$y - mean(datrain$y))^2) # Somme des carrés totaux ss_residual <- sum((datrain$y -
  rf_model$predictions)^2) # Somme des carrés résiduels r_squared_oob <- 1 - (ss_residual / ss_total)
  return(list(Score = r_squared_oob, Pred = 0))
}

# Optimisation bayésienne des hyper-paramètres # (mtry,
min.node.size et sample.fraction)
opt_results <- BayesianOptimization(FUN = rf_optimization,
                                    bounds = list(mtry = c(2, 10), min_node_size = c(1, 10),
                                                  sample_fraction = c(0.5, 1)),
                                    init_points = 5, # Nombre de points initiaux n_iter = 15, # Nombre
                                    d'itérations
                                    kappa = 2.576, # Exploration-exploitation trade-off verbose = FALSE)
```

```
##
## Best Parameters Found:
  ## Round = 12                mtry = 9.304423 min_node_size = 1.0000 sample_fraction = 1.0000          Value =
0.8004113
```

Bibliographie

1. Galar, M., Fernández, A., Barrenechea, E., Bustince, H., & Herrera, F. (2011). « An overview of ensemble methods for binary classifiers in multi-class problems: Experimental study on one-vs-one and one-vs-all schemes ». Pattern Recognition, 44(8), 1761-1776.

2. Yang, Li & Abdallah Shami (2020, 20 novembre). « On hyperparameter optimization of machine learning algorithms: Theory and practice », *Neurocomputing*, vol. 415. Récupéré de https: //doi.org/10.1016/j.neucom.2020.07.061

3. Yu, Tong & Hong Zhu (2020, 12 mars). « Hyper-Parameter Optimization: A Review of Algorithms and Applications », *CoRR*, vol. abs/2003.05689. Récupéré de https://arxiv.org/abs/2003.0 5689

4. Microsoft (2024). *Optimisation des hyperparamètres d'un modèle (v2)*, Microsoft Learn. Récupéré de https://learn.microsoft.com/fr-fr/azure/machine-learning/how-to-tune-hyperpa rameters?view=azureml-api-2

5. Scikit-learn (2024)). *Multiclass and multioutput algorithms*, Scikit Documentation. Récupéré de https://scikit-learn.org/stable/modules/multiclass.html#multiclass-and-multioutp ut-algorithms

6. FLAML (2024)). *Getting Started*, FLAML Documentation. Récupéré de https://microsoft. github.io/FLAML/docs/Getting-Started