Projet PL : Modèle Random Forest

1 Introduction

Le modèle **Random Forest** est utilisé pour identifier différents types d'apprenants (visuels, auditifs, kinesthésiques) à partir de leurs comportements et performances scolaires. L'objectif est de personnaliser les parcours d'apprentissage en fonction des préférences individuelles, afin d'améliorer l'efficacité pédagogique. En analysant des caractéristiques comme les préférences d'étude, les scores académiques, et les habitudes d'apprentissage, ce modèle permet de prédire le type d'apprenant et de proposer des méthodes adaptées.

2 Fonctionnement du modèle Random Forest

Le modèle **Random Forest** est constitué de plusieurs arbres de décision, chacun construit à partir de sous-échantillons aléatoires des données. Chaque arbre effectue une prédiction, et la classe finale est déterminée par un vote majoritaire des arbres, ce qui permet de réduire le sur-apprentissage (overfitting) et d'améliorer la précision des prédictions.

3 Pseudo-code pour la construction d'un arbre de décision et de la forêt aléatoire

3.1 Classe DecisionTree

La classe **DecisionTree** construit un arbre de décision en divisant les données à chaque niveau en fonction des caractéristiques les plus informatives, en utilisant l'entropie pour évaluer les splits.

```
class DecisionTree:
    def __init__(self, max_depth=None):
        self.max_depth = max_depth
        self.tree = None
    def fit(self, X, y):
        self.tree = self._build_tree(X, y, depth=0)
    def predict(self, X):
        predictions = []
        for row in X:
            prediction = self._predict_row(row, self.tree)
            predictions.append(prediction)
        return predictions
    def _build_tree(self, X, y, depth):
        if len(set(y)) == 1 or depth >= self.max_depth:
            return self._majority_class(y)
        feature, threshold = self._best_split(X, y)
        if feature is None:
            return self._majority_class(y)
        left_indices = X[:, feature] <= threshold
        right_indices = ~left_indices
        left_subtree = self._build_tree(X[left_indices], y[
           left_indices], depth + 1)
        right_subtree = self._build_tree(X[right_indices], y[
           right_indices], depth + 1)
        return {"feature": feature, "threshold": threshold, "left":
           left_subtree, "right": right_subtree}
```

3.2 Classe RandomForest

Le modèle **RandomForest** combine plusieurs arbres de décision. Chaque arbre est formé à partir d'un sous-ensemble aléatoire des données d'entraînement (bootstrap). Une fois tous les arbres entraînés, la classe finale pour une observation est déterminée par un vote majoritaire.

```
class RandomForest:
    def __init__(self,n_trees=10,max_depth=None, sample_size=None):
        self.n_trees = n_trees
        self.max_depth = max_depth
        self.sample_size = sample_size
        self.trees = []
    def fit(self, X, y):
        for _ in range(self.n_trees):
            indices = np.random.choice(len(X), size=self.sample_size
               , replace=True)
            X_sample = X[indices]
            y_sample = y[indices]
            tree = DecisionTree(max_depth=self.max_depth)
            tree.fit(X_sample, y_sample)
            self.trees.append(tree)
    def predict(self, X):
        tree_predictions = []
        for tree in self.trees:
            predictions = tree.predict(X)
            tree_predictions.append(predictions)
        return self._majority_vote(tree_predictions)
    def _majority_vote(self, predictions):
        return [max(set(pred), key=pred.count) for pred in zip(*
           predictions)]
```

4 Complexité temporelle

La complexité temporelle des différentes méthodes est résumée dans le tableau suivant :

Opération/Méthode	Description du Pseudo-code	Complexité temporelle
Arbre de Décision		
Constructeur	Initialisation de max_depth et arbre vide	$O(1)$
fit	Construction de l'arbre à partir des données d'entraînement	$O(dn^2 \log n)$ où: - d = nombre de caractéristiques - n = nombre d'échantillons
predict (individuel)	Navigation dans l'arbre de la racine à la feuille	$O(\log n)$
predict (tous)	Prédiction pour tous les échantillons	$O(m \log n)$ où m est le nombre d'échantillons
construire_arbre	Construction récursive de la structure de l'arbre	$O(dn^2 \log n)$
trouver_meilleur_split	Recherche de la meilleure caractéristique et seuil	$O(dn^2)$ où: - d = nombre de caractéristiques - n = nombre d'échantillons
calculer_gain_information	Calcul du gain d'information pour la division	$O(n)$
calculer_entropie	Calcul de l'entropie du sous-ensemble	$O(n)$
prédire_ligne	Parcours de l'arbre pour une prédiction	$O(\log n)$
visualize_tree	Affichage récursif de l'arbre complet	$O(2^{\log n}) = O(n)$
Forêt Aléatoire		
Constructeur	Initialisation des paramètres	$O(1)$
fit	Construction de multiples arbres avec échantillons bootstrap	$O(t \cdot (n + dn^2 \log n))$ où: - t = nombre d'arbres - n = taille d'échantillon - d = nombre de caractéristiques
predict	Obtention des prédictions de tous les arbres et vote	$O(t \cdot m \log n)$ où: - t = nombre d'arbres - m = nombre d'échantillons à prédire
vote_majoritaire	Comptage de la prédiction la plus fréquente	$O(t \cdot m)$

Table 1: Analyse de la complexité des algorithmes d'Arbre de Décision et Forêt Aléatoire

Complexité totale du modèle :

- **Temps d'entraînement** : $O(t \cdot d \cdot n^2 \log n)$
- **Temps de prédiction** : $O(t \cdot m \log n)$

Comparaison des Performances des Langages Python, R et Julia *from scratch*

Objectif

Cette étude compare l'implémentation *from scratch* du modèle Random Forest en Python, R et Julia. Nous mesurons le temps d'exécution et la précision du modèle pour chaque langage, tout en abordant les difficultés liées à l'implémentation sans recourir à des bibliothèques spécialisées.

Méthodologie

Le modèle Random Forest a été implémenté *from scratch* dans chaque langage :

- **Python** : Implémentation manuelle avec des structures de données de base.
- **R** : Implémentation manuelle avec des fonctions de base.
- **Julia** : Implémentation manuelle sans bibliothèques tierces, utilisant des fonctions natives de Julia.

Nous avons mesuré le temps d'exécution et la précision pour différentes configurations des paramètres du modèle, tels que la profondeur des arbres, le nombre d'arbres et la taille des échantillons.

Résultats

Langage	Depth	n_trees	Execution Time (s)	Accuracy
Python	3	5	1.92	0.64
Python	5	10	5.30	0.72
Python	7	15	9.70	0.75
R	3	5	2.10	0.60
R	5	10	11.50	0.68
R	7	15	15.20	0.72
Julia	3	5	1.85	0.62
Julia	5	10	4.80	0.70
Julia	7	15	6.50	0.74

Table 1: Comparaison des performances des modèles Random Forest implémentés *from scratch* dans différents langages.

Difficultés Rencontrées

- **Python** : L'implémentation *from scratch* nécessite la gestion manuelle de structures comme les arbres de décision. Bien que Python soit flexible, cela a rendu l'optimisation plus difficile.
- **R** : En raison de l'absence de bibliothèques spécialisées, l'implémentation manuelle en R a exigé beaucoup de code personnalisé, ce qui a augmenté la complexité du développement.
- **Julia** : L'implémentation en Julia, bien que performante, a nécessité un certain travail supplémentaire pour gérer les structures de données et les arbres de décision manuellement. Cependant, le langage s'est révélé performant.

Conclusion

Dans cette étude, bien que l'implémentation *from scratch* ait été un défi dans tous les langages, Python a été préféré en raison de sa simplicité pour gérer les structures de données et la clarté de son code. Julia a montré des performances intéressantes, mais a nécessité un plus grand effort d'implémentation, tandis que R a été plus difficile à manipuler sans bibliothèques externes.

Remarque

En termes de vitesse d'exécution, Julia est le plus rapide, suivi de Python, tandis que R est le plus lent.