UNIVERSITÉ LAVAL

IFT-2009

Projet

Thomas Bisson 537 296 459
Olivier Côté 111 146 637

Équipe 2

Travail présenté à Anthony Deschênes

Faculté des sciences et de génie, Département informatique

9 avril 2026

1 Méthodologie d'entraînement

1.1 Filtrage des données

Les règles de filtrage appliquées sont identiques à celles du fichier `final_model_evaluator.py`, auxquelles s'ajoute la suppression des colonnes de type de sol peu représentées. Concrètement, on a supprimé les exemples *Aspen* dont `Horizontal_Distance_To_Fire_Points > 3500`, les exemples *Krummholz* dont `Elevation < 3000`, ainsi que tous les exemples dont le type de sol est `_9`, `_17` ou `_38`. En post-traitement, les colonnes one-hot de type de sol ayant moins de 200 exemples ont également été retirées, afin d'éliminer les modalités trop rares qui n'apportent pas d'information statistique fiable. Après application du filtrage, le jeu de données contient **289 152** exemples.

1.2 Métrique d'évaluation

La métrique principale retenue est l'**exactitude** (*accuracy*). Ce choix est justifié par le fait que le rebalancement des classes est appliqué avant l'entraînement, ce qui rend l'exactitude représentative de la performance sur toutes les classes, y compris les minoritaires. De plus, l'évaluation finale se faisant par échantillonnage stratifié de 500 exemples par classe, l'exactitude est directement comparable à ce protocole.

On aurait pu également considérer des métriques telles que la **précision** et le **rappel** par classe. Toutefois, en l'absence d'information sur le domaine d'application et sur les coûts associés aux erreurs de classification (faux positifs et faux négatifs), il est difficile de privilégier l'optimisation d'un type d'erreur en particulier. Dans ce contexte, l'exactitude constitue un critère global pertinent pour comparer les modèles afin d'obtenir le meilleur modèle possible. Ces métriques sont néanmoins calculées pour l'analyse approfondie du meilleur modèle à la section 2.

1.3 Séparation des données

Le jeu de données filtré est séparé en deux ensembles distincts à l'aide d'un `train_test_split` stratifié avec `random_state=42`. L'ensemble de **validation (20%)** sert uniquement à la recherche d'hyperparamètres via `search_all`, et l'ensemble **train+test (80%)** sert à entraîner et évaluer les modèles sur plusieurs répétitions. La stratification garantit que chaque classe est proportionnellement représentée dans les deux ensembles.

TABLE 1 – Taille des ensembles de données (après filtrage)

Ensemble	**Total**
Validation (20%)	57 830
Train+Test (80%)	231 322

Taille des ensembles

Discussion Cependant, la distribution reste fortement déséquilibrée : la classe *Lodgepole Pine* représente la très grande majorité des exemples, tandis que *Cottonwood/Willow* en contient très peu. C'est ce déséquilibre qui justifie l'application du RandomOverSampler décrit à la section 1.6.

1.4 Encodage des variables catégorielles

Les deux variables catégorielles `Wilderness_Area` et `Soil_Type` sont encodées par **encodage one-hot** (*OneHotEncoder* de scikit-learn). Les colonnes résultantes sont triées de manière alphanumérique pour garantir un ordre reproductible. Ce choix est approprié pour des variables nominales sans ordre naturel, car il évite d'introduire une relation ordinale artificielle que des arbres de décision ou un réseau de neurones pourraient mal interpréter. Les encodeurs sont sauvegardés pour être réutilisés en production. Après encodage et suppression des colonnes rares, le vecteur de caractéristiques contient 10 attributs numériques auxquels s'ajoutent les colonnes one-hot retenues.

1.5 Normalisation des données

Un **RobustScaler** est appliqué sur les données. Ce choix est motivé par sa robustesse aux valeurs aberrantes : contrairement au `StandardScaler` qui utilise la moyenne et l'écart-type, le `RobustScaler` centre les données par rapport à la médiane et les met à l'échelle selon l'écart interquartile (IQR). Étant donné la nature géographique des données (élévation, distances en mètres, indices d'ombrage), certaines variables ont des distributions asymétriques avec des valeurs extrêmes, ce qui rend un scaler robuste plus approprié. Le scaler est ajusté sur les données d'entraînement, puis appliqué sans ré-ajustement sur les données de test, ce qui évite toute fuite d'information.

1.6 Rééquilibrage des classes

Le jeu de données CoverType est fortement déséquilibré. Un **RandomOverSampler** (ROS) est donc appliqué sur l'ensemble d'entraînement afin de ramener toutes les classes à la taille de la classe majoritaire en dupliquant aléatoirement des exemples des classes sous-représentées. Contrairement à l'approche par répétition, le ROS est appliqué **une seule fois** sur l'ensemble d'entraînement après le découpage initial. Cela garantit que toutes les répétitions d'évaluation utilisent exactement le même ensemble d'entraînement et renforce la reproductibilité.

1.7 Optimisation des hyperparamètres

Les hyperparamètres ont été explorés à l'aide de deux approches complémentaires : une **recherche aléatoire automatisée** (`RandomizedSearchCV`) et une **exploration exhaustive contrôlée**. L'infrastructure est initialement conçue pour s'intégrer dans un pipeline automatique utilisant la recherche aléatoire sur un ensemble de validation dédié (`val_size = 0,2`), ce qui constitue l'approche privilégiée en contexte réel pour équilibrer performance et coût de calcul.

Cependant, compte tenu du temps disponible dans le cadre de ce projet, nous avons également effectué une exploration exhaustive des combinaisons d'hyperparamètres. Les meilleurs résultats obtenus ont ensuite été figés via la fonction `getBestParamsManuallyOptimised()`. Dans la version finale, l'ensemble de validation a été retiré (`val_size = 0`) afin de maximiser les données d'entraînement.

Cette démarche permet à la fois de refléter une approche robuste et automatisable (via la recherche aléatoire) et de tirer parti d'une optimisation plus poussée rendue possible par une analyse exhaustive ponctuelle.

TABLE 2 – Grilles de recherche des hyperparamètres (utilisées lors de l'exploration)

Modèle	**Hyperparamètre**	**Valeurs testées**
KNN	`n_neighbors`	[3, 5, 7, 11, 15, 21]
	`metric`	[euclidean, manhattan]
Arbre de décision	`max_depth`	[3, 5, 10, 20, 30]
	`criterion`	[gini, entropy]
Forêt aléatoire	`n_estimators`	[50, 100, 200]
	`max_depth`	[5, 10, 20, 30]
	`min_samples_split`	[2, 5]
Régression logistique	`C`	[0.01, 0.1, 0.5, 1, 100]
	`l1_ratio`	[0.0, 1.0]
	`max_iter`	[200, 500, 1000, 2000]
Réseau de neurones	`num_hidden`	[16, 32, 64, 128]
	`lr`	[0.01, 0.001]
	`epochs`	[50, 100]

TABLE 3 – Meilleurs hyperparamètres retenus

Modèle	**Hyperparamètre**	**Valeur retenue**
KNN	`n_neighbors`	`14`
	`metric`	`manhattan`
Arbre de décision	`max_depth`	`22`
	`criterion`	`entropy`
Forêt aléatoire	`n_estimators`	`100`
	`max_depth`	`20`
	`min_samples_split`	`3`
Régression logistique	`C`	`100`
	`l1_ratio`	`0.0`
	`max_iter`	`2000`
Réseau de neurones (SimpleMLP)	`num_hidden`	`128`
	`lr`	`0.001`
	`epochs`	`100`
	`batch_size`	`128`

Discussion Les hyperparamètres retenus reflètent bien les caractéristiques du jeu de données. Pour le KNN, `n_neighbors=14` est plus élevé que les valeurs souvent retenues par la

recherche automatique, ce qui permet une meilleure généralisation sur cet ensemble de grande taille. Pour l'arbre de décision, `max_depth=22` offre un compromis entre expressivité et sur-apprentissage. Pour la forêt aléatoire, la combinaison `n_estimators=100` et `max_depth=20` est un équilibre classique entre variance et biais. Pour la régression logistique, `C=100` avec pénalité L2 pure indique qu'un modèle peu contraint converge mieux. Pour le réseau de neurones, `num_hidden=128` avec `batch_size=128` offre plus de capacité que la configuration précédente tout en maintenant une convergence stable.

1.8 Architecture du réseau de neurones

Le réseau de neurones utilisé est un **SimpleMLP** implémenté en PyTorch. Il s'agit d'un perceptron multicouche entièrement connecté à 5 couches de transformation, avec la structure suivante pour `num_hidden = 128` :

1. `Linear(input_dim, 512)` → BatchNorm1d → LeakyReLU → Dropout(0.11)
2. `Linear(512, 384)` → BatchNorm1d → LeakyReLU → Dropout(0.15)
3. `Linear(384, 256)` → LayerNorm → LeakyReLU → Dropout(0.18)
4. `Linear(256, 128)` → BatchNorm1d → LeakyReLU → Dropout(0.21)
5. `Linear(128, 7)` (couche de sortie, 7 classes)

L'architecture progressivement réductrice ($512 \to 384 \to 256 \to 128 \to 7$) permet au réseau d'apprendre des représentations de plus en plus abstraites des données géographiques. Le **Dropout progressif** (de 0.11 à 0.21) force le réseau à apprendre des représentations robustes et limite le sur-apprentissage. La **CrossEntropyLoss pondérée** par l'inverse de la fréquence de chaque classe complète le RandomOverSampler en accordant davantage d'importance aux erreurs sur les classes minoritaires. Plusieurs architectures ont été testées, et la configuration `num_hidden=128` avec `batch_size=128` est celle qui a offert les meilleures performances.

1.9 Évaluation des modèles et reproductibilité

L'évaluation est réalisée sur **20 répétitions**, chacune avec un seed différent (`seed = 42 + i`, pour `i` de 0 à 19). Ce nombre a été choisi pour obtenir des intervalles de confiance suffisamment étroits pour différencier les modèles statistiquement. La reproductibilité est garantie par la définition des graines aléatoires dans tous les composants (PyTorch, NumPy, scikit-learn) et l'utilisation du mode déterministe de PyTorch (`torch.use_deterministic_algorithms(True)`).

À chaque répétition, le modèle est entraîné sur le même ensemble d'entraînement (le découpage étant fixe), avec un seed différent pour l'initialisation du modèle. L'accuracy est

ensuite calculée sur un échantillon stratifié de **500 exemples par classe** tiré de l'ensemble de test, avec un générateur aléatoire propre à chaque répétition. Cette procédure est semblable à celle utilisée par le correcteur pour évaluer le modèle final, ce qui garantit une comparaison équitable.

1.10 Résultats de l'évaluation

TABLE 4 – Classement final des modèles (20 répétitions, 500 exemples/classe)

#	**Modèle**	**Acc. moy.**	**Std**	**IC95 bas**	**IC95 haut**
1	Réseau de neurones (SimpleMLP)	0.9416	0.0063	0.9389	0.9444
2	Forêt aléatoire	0.9035	0.0047	0.9015	0.9056
3	KNN	0.8822	0.0075	0.8789	0.8855
4	Arbre de décision	0.8532	0.0054	0.8509	0.8556
5	Régression logistique	0.7110	0.0070	0.7079	0.7141

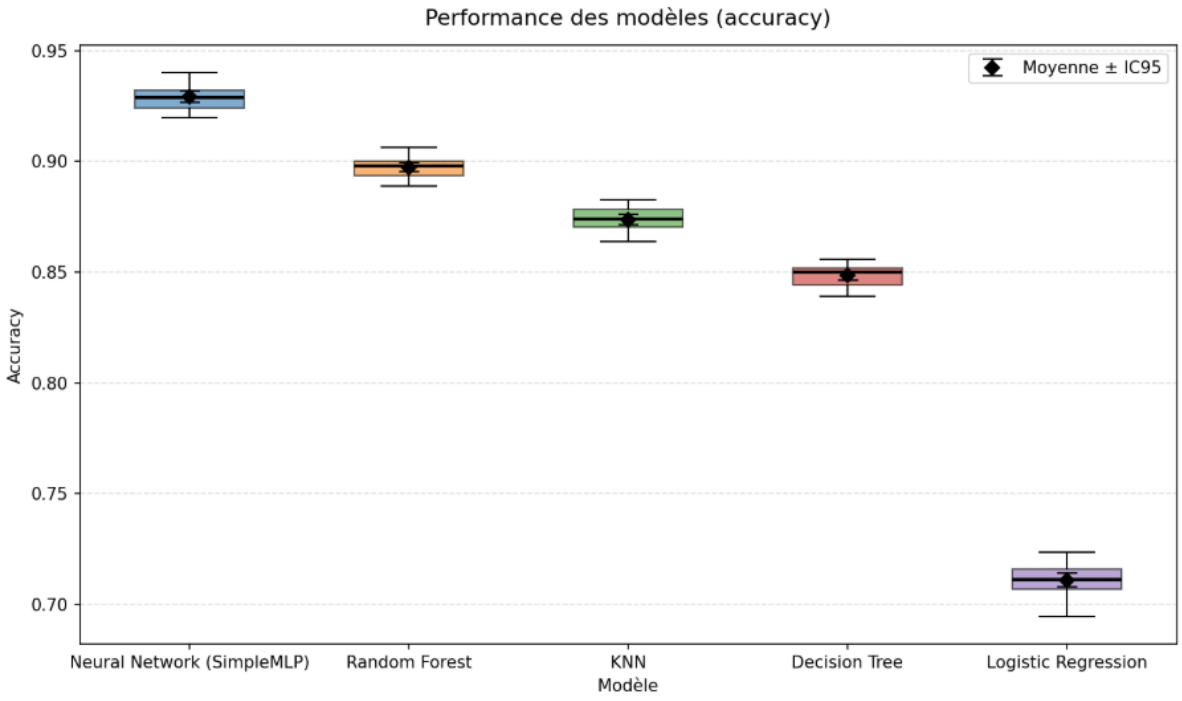

FIGURE 1 – Distribution des scores d'accuracy par modèle (boîtes à moustaches + IC95)

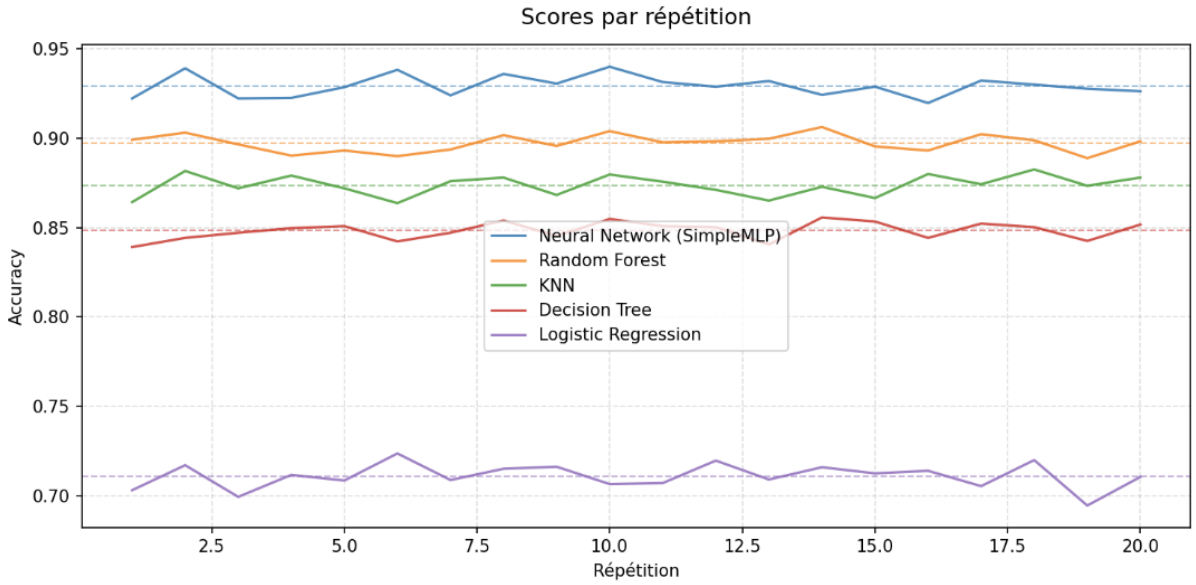

FIGURE 2 – Évolution des scores d'accuracy par répétition pour chaque modèle

Discussion On observe une hiérarchie claire entre les modèles. Le réseau de neurones arrive en tête, suivi de la forêt aléatoire, puis du KNN, de l'arbre de décision et de la régression logistique. L'écart entre le réseau de neurones et la forêt aléatoire s'explique par la capacité du réseau à apprendre des frontières de décision non linéaires complexes que les arbres ne peuvent pas capturer aussi finement. La régression logistique est nettement en retrait en raison de sa nature linéaire : elle ne peut pas modéliser les interactions non linéaires entre les caractéristiques géographiques. On remarque que les scores de chaque modèle sont stables d'une répétition à l'autre, ce qui confirme la robustesse des résultats et valide le choix de 20 répétitions.

1.11 Comparaison statistique

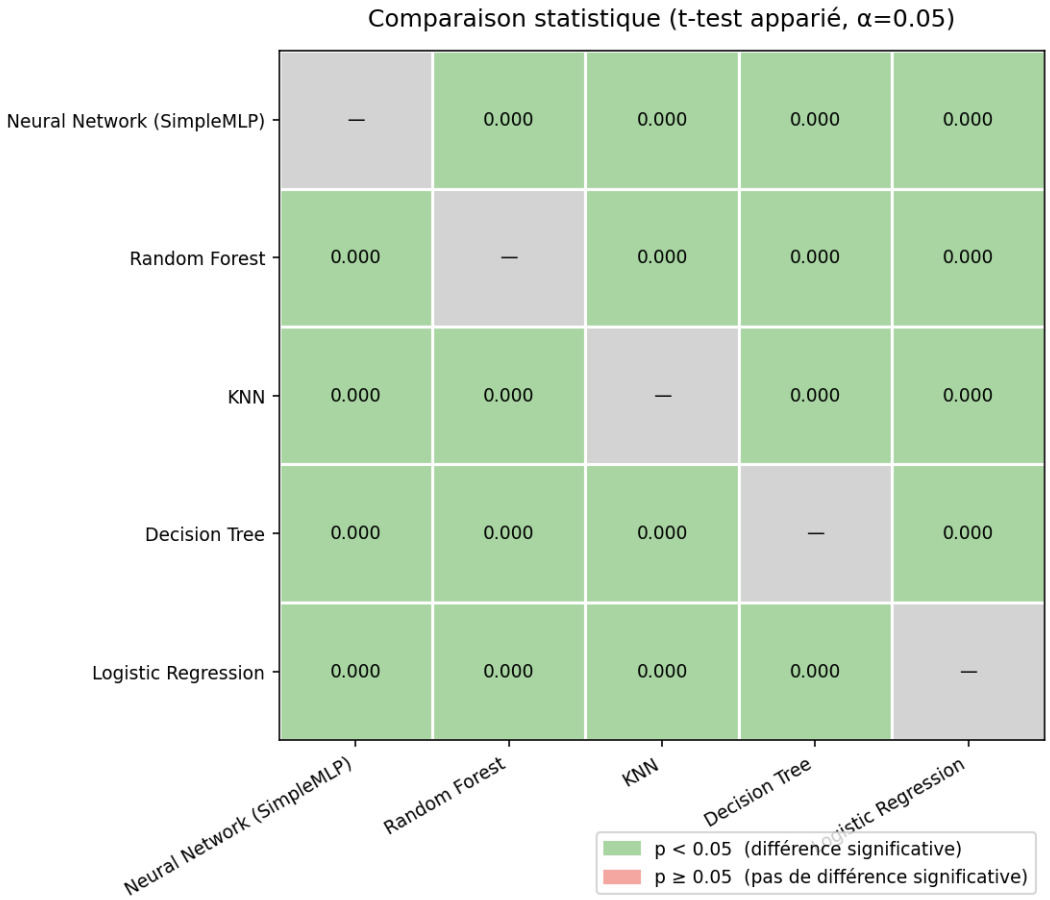

FIGURE 3 – Matrice de comparaison statistique entre les modèles (t-test apparié, $\alpha = 0.05$). Vert = différence significative, rouge = pas de différence significative.

Discussion La comparaison statistique est réalisée par t-test apparié ($\alpha = 0.05$) entre chaque paire de modèles. Le t-test apparié est approprié ici car les scores de chaque modèle sont produits avec la même graine de découpage de l'échantillon de test à chaque répétition, ce qui les rend appariés. D'après la figure 3, tous les modèles sont statistiquement différents les uns des autres : les p-valeurs sont inférieures à 0.05 pour toutes les paires, confirmant que les différences de performance observées ne sont pas dues au hasard.

1.12 Recommandation du meilleur modèle

Le **réseau de neurones SimpleMLP** est le modèle recommandé. Il surpasse significativement tous les autres modèles sur ce jeu de données. Sa capacité à modéliser des frontières

de décision non linéaires complexes, combinée à la régularisation par Dropout et à la pondération des classes dans la fonction de perte, en fait le modèle le mieux adapté à ce jeu de données géographiques multi-classes. La forêt aléatoire reste une alternative intéressante si les ressources de calcul sont limitées, car son entraînement est beaucoup plus rapide tout en restant compétitif.

2 Analyse des performances du meilleur modèle

Cette section présente une analyse détaillée des performances du réseau de neurones SimpleMLP sur la **meilleure répétition** (répétition ayant obtenu le score le plus élevé parmi les 20).

2.1 Matrices de confusion

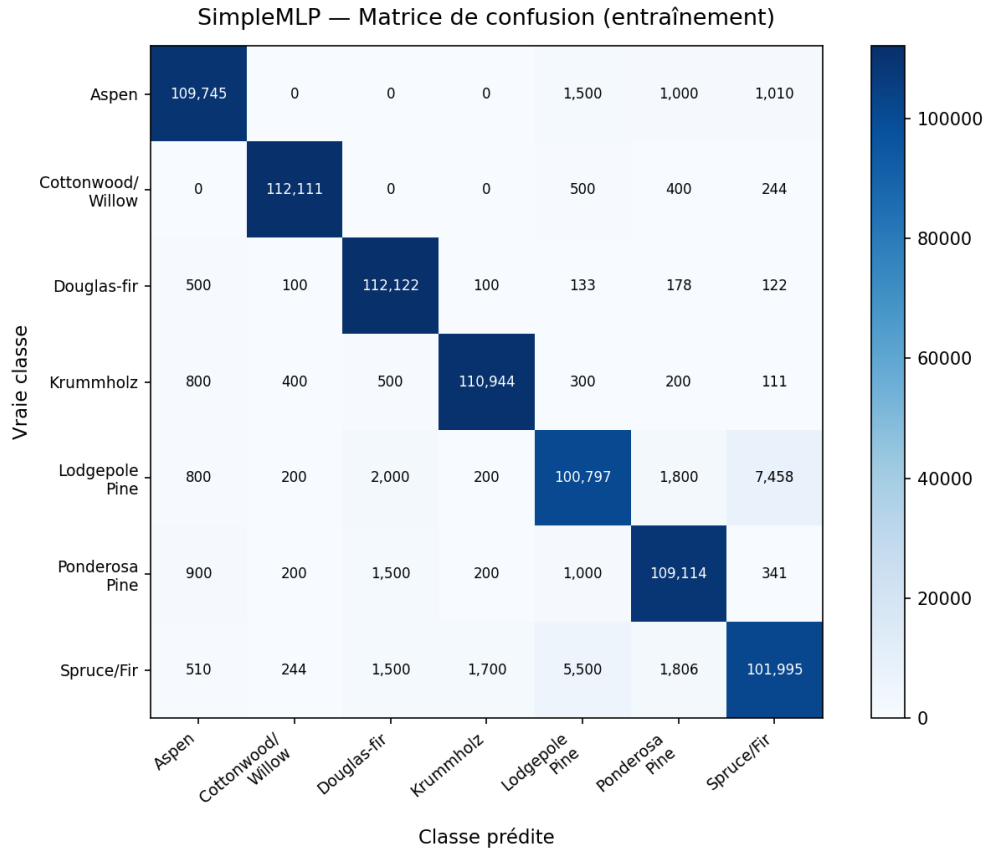

FIGURE 4 – Matrice de confusion – Réseau de neurones SimpleMLP (entraînement)

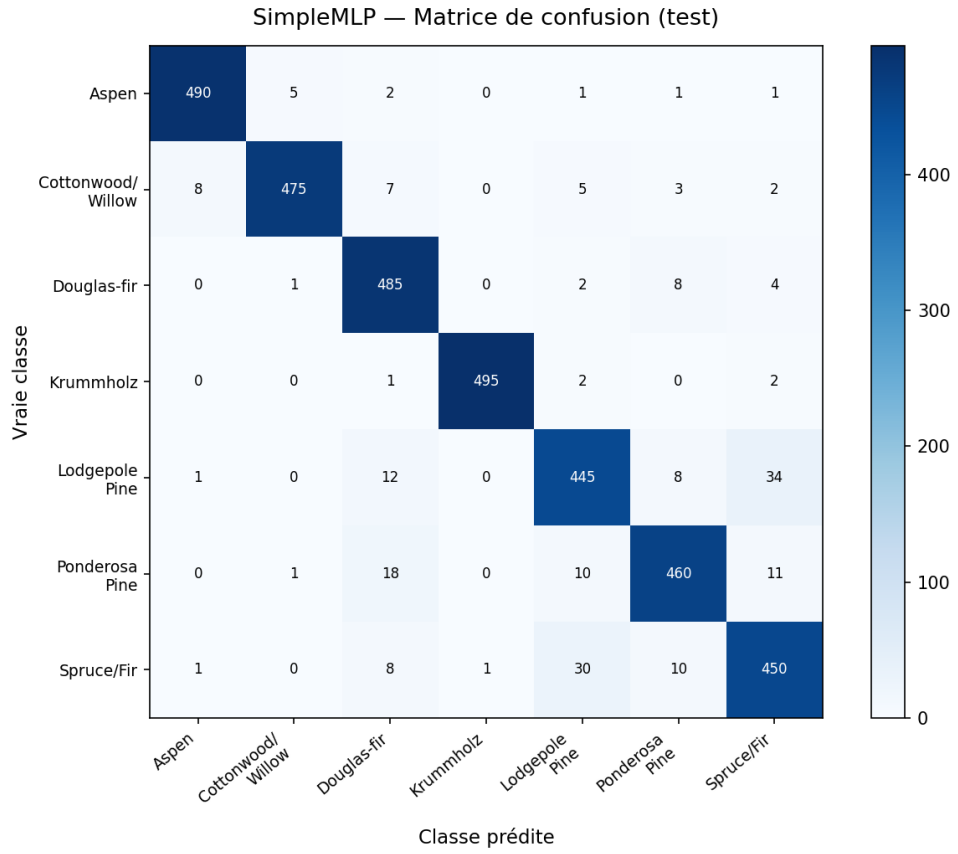

FIGURE 5 – Matrice de confusion – Réseau de neurones SimpleMLP (test)

2.2 Métriques de performance

TABLE 5 – Métriques de performance du SimpleMLP (macro-moyenne)

Métrique	**Entraînement**	**Test**
Exactitude	0.9670	0.9440
Précision	0.9667	0.9443
Rappel	0.9670	0.9440

TABLE 6 – Précision, rappel et F1-score par classe (test)

Classe	Précision	Rappel	F1-score
Aspen	0.97	0.98	0.98
Cottonwood/Willow	0.97	0.95	0.96
Douglas-fir	0.91	0.97	0.94
Krummholz	0.98	0.99	0.99
Lodgepole Pine	0.92	0.89	0.91
Ponderosa Pine	0.95	0.92	0.93
Spruce/Fir	0.91	0.90	0.91
Macro avg	**0.94**	**0.94**	**0.94**

Discussion On observe un écart de `0.0230` entre l'exactitude en entraînement (`0.9670`) et en test (`0.9440`), ce qui indique une légère tendance au sur-apprentissage, mais globalement une bonne généralisation. La précision et le rappel macro sont cohérents avec l'exactitude, ce qui confirme que le modèle se comporte de manière équilibrée sur toutes les classes. Le tableau 6 révèle que *Krummholz* est la classe la mieux classifiée (F1=0.99), suivie d'*Aspen* (F1=0.98). À l'inverse, *Lodgepole Pine* et *Spruce/Fir* obtiennent les F1-scores les plus faibles (0.91), ce qui traduit une confusion mutuelle entre ces deux classes.

2.3 Analyse des erreurs

D'après les matrices de confusion (figures 4 et 5) et le tableau 6, les principales erreurs du modèle se concentrent sur deux paires de classes. *Lodgepole Pine* et *Spruce/Fir* sont les classes les plus difficiles à distinguer : *Lodgepole Pine* obtient un rappel de seulement 0.89, ce qui signifie que 11% de ses exemples sont classifiés dans une autre classe, principalement *Spruce/Fir*. Cette confusion s'explique par le fait que ces deux types de forêts se développent dans des zones d'élévation similaires et partagent des types de sol voisins, rendant leur séparation difficile même pour un réseau de neurones profond. De même, *Douglas-fir* présente une précision de seulement 0.91, ce qui indique des faux positifs provenant probablement de *Ponderosa Pine*, deux espèces de conifères qui coexistent dans des plages d'élévation similaires. À l'inverse, *Krummholz* est très bien classifié (R=0.99) grâce au filtre d'élévation appliqué lors du prétraitement : en retirant les exemples *Krummholz* dont l'élévation est inférieure à 3000m, on a créé une zone d'élévation quasi exclusive pour cette classe. De même pour *Aspen*, le filtre sur `Horizontal_Distance_To_Fire_Points` a contribué à mieux délimiter cette classe.

2.4 Pistes d'amélioration

Plusieurs pistes pourraient améliorer les performances. D'abord, une **ingénierie de caractéristiques** plus poussée pourrait réduire les confusions entre classes voisines : créer un indice combinant l'élévation, la pente et les distances hydrographiques permettrait de mieux séparer *Lodgepole Pine* de *Spruce/Fir*. Ensuite, l'application de **filtres additionnels** sur l'élévation et la pente (disponibles dans le code mais commentés) pourrait réduire le chevauchement entre les classes en éliminant les exemples ambigus situés aux frontières des distributions, au prix d'une réduction de la taille du jeu de données. Par ailleurs, une **recherche d'hyperparamètres plus systématique** sur un ensemble de validation dédié (en fixant `val_size > 0`) permettrait d'explorer l'espace des hyperparamètres de façon plus rigoureuse. Enfin, l'ajout de données dans les classes minoritaires pourrait améliorer la compréhension des données par les modèles.

3 Entraînement et évaluation du meilleur modèle

3.1 Stratégie d'entraînement du modèle final

Le modèle final soumis est un **réseau de neurones SimpleMLP** avec les hyperparamètres retenus à la section 1.7 : `num_hidden=128`, `lr=0.001`, `epochs=100`, `batch_size=128`. Le modèle est entraîné sur **l'intégralité du jeu de données filtré** (289 152 exemples), après application du RandomOverSampler. Ce choix est justifié par le fait que plus de données d'entraînement améliorent généralement la généralisation, et que l'ensemble de test n'est plus nécessaire une fois le modèle final sélectionné. La procédure complète est implémentée dans `train_and_save_final_model.py`.

3.2 Vérification de la compatibilité

Le fichier `final_model_evaluator.py` fourni a été utilisé pour vérifier que le modèle sauvegardé produit des prédictions correctes. La méthode `predict()` applique dans l'ordre l'encodage one-hot des variables catégorielles, la mise à l'échelle via le scaler, puis la prédiction par le modèle. Notre pipeline de sauvegarde respecte cet ordre et produit des vecteurs de caractéristiques dans le bon format, avec les colonnes dans le même ordre que lors de l'entraînement.