

**Marius Casamian**  
**Cm205773**  
**L3 IA Projet 2024**  
**10/12/2024 pour le 16/11/2024**

## **Bilan sujet 1 :Prédiction de l'approbation de prêt**

Données : Informations sur les demandeurs de prêt  
Prédire si un demandeur de prêt est approuvé. (loan\_status = 1)  
Type de problème : Classification binaire {0,1}

Librairie utilisée pour le problème: matplotlib, seaborn, torch, pandas, numpy  
target : loan\_status

Choix modèles: à définir

Méthode d'implémentation: Python et notebook, NEF

Ressources : [https://en.wikipedia.org/wiki/Receiver\\_operating\\_characteristic](https://en.wikipedia.org/wiki/Receiver_operating_characteristic)

Documentations: Medium, Youtube, Kaggle, Internet

Plan du projet:

1. Importer le dataset train.csv et tenter de le comprendre, EDA (exploratory data analysis).Comprendre graphiquement et statistiquement les corrélations entre les features, ex: home ownership avec loan\_status etc.
2. Étapes de pré-processing: outliers, normalisation, équilibre des classes
3. Pipeline, modèles, métriques AUC, entraînements
4. Comparaisons et figures de convergences
5. Hyper paramètres tuning
6. Soumission Kaggle et NEF

Ce plan basique est censé répondre aux exigences demandées pour ce problème et par les consignes.

Nous allons expliquer étapes par étapes, grâce au code Python du notebook comment y parvenir.

## I) EDA

L'objectif de départ est de comprendre le jeu de données. On sait qu'il s'agit d'un problème de classification binaire, c'est-à-dire dont le but est de prédire si un prêt est accepté ( $y=1$ ) ou refusé ( $y=0$ ).

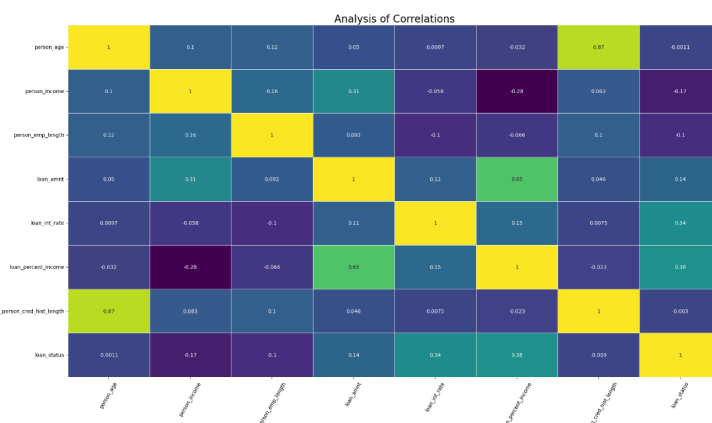
On va commencer par importer le dataset de train.csv qui constitue nos données d'entraînement, là où nos modèles effectuent des entraînements avant de les généraliser à de nouvelles données.

Je regarde si mon dataset possède des valeurs manquantes, ici il n'y en a pas, je regarde ensuite les valeurs uniques,  $\Leftrightarrow$  différentes pour chaque feature, ceci me permet de mieux comprendre pour mieux traiter.

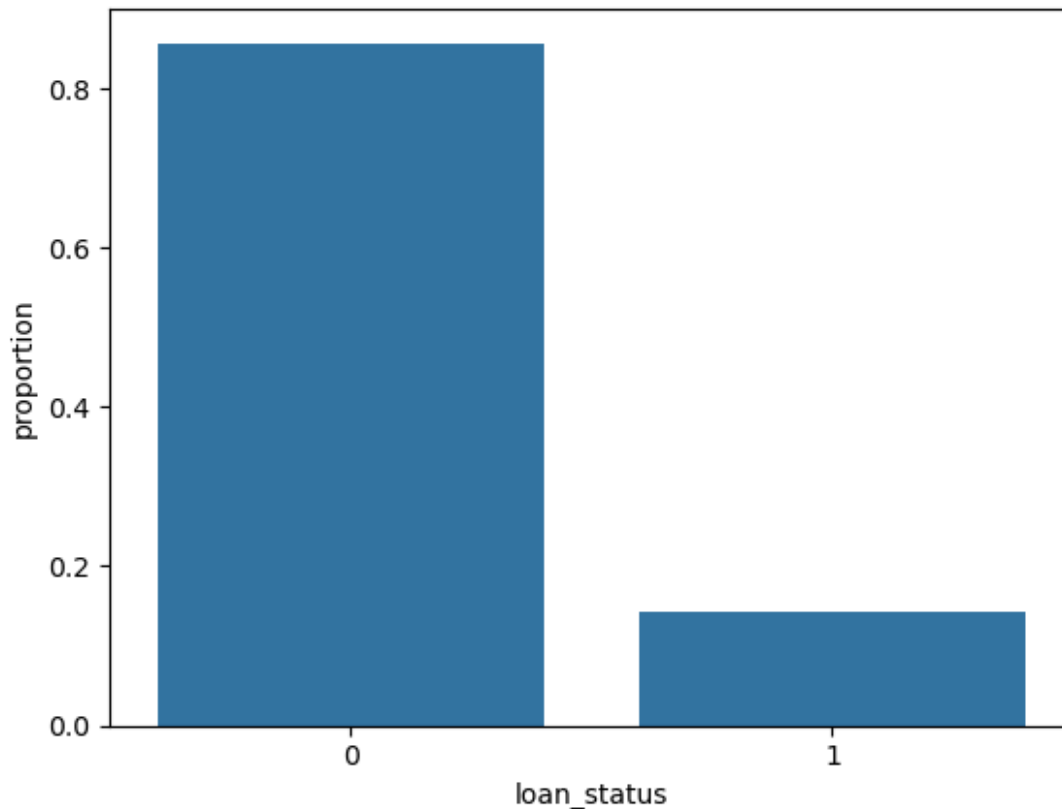
	Number of Unique Values	Unique Values
cb_person_cred_hist_length	29	[14, 2, 10, 5, 3, 11, 6, 9, 7, 8, 4, 17, 13, 1...
cb_person_default_on_file	2	[N, Y]
loan_amnt	545	[6000, 4000, 12000, 9000, 2500, 5000, 15000, 1...
loan_grade	7	[B, C, A, D, E, F, G]
loan_int_rate	362	[11.49, 13.35, 8.9, 11.11, 6.92, 8.94, 6.54, 1...
loan_intent	6	[EDUCATION, MEDICAL, PERSONAL, VENTURE, DEBTCO...
loan_percent_income	61	[0.17, 0.07, 0.21, 0.1, 0.2, 0.27, 0.13, 0.14, ...]
loan_status	2	[0, 1]
person_age	53	[37, 22, 29, 30, 27, 25, 21, 35, 31, 26, 28, 3...
person_emp_length	36	[0.0, 6.0, 8.0, 14.0, 2.0, 9.0, 11.0, 5.0, 1.0, ...]
person_home_ownership	4	[RENT, OWN, MORTGAGE, OTHER]
person_income	2641	[35000, 56000, 28800, 70000, 60000, 45000, 200...

Cette image nous le montre, on voit la target [0,1] et des features que je vais devoir encoder plus tard.

Ensuite je regarde les corrélations entre les features pour voir si je peux en supprimer car trop fortement corrélé, ou au contraire voir si des variables sont intéressantes pour l'analyse. (cf notebook)



On peut voir une forte corrélation positive entre `person_age` et `cb_person_cred_hist_length` ou encore `loan_amnt` et `loan_percent_income`. Ensuite je décide de regarder la proportion de ma target.



On peut voir qu'elle est fortement mal équilibrée, ce qui peut poser problème car on souhaite prédire, la variable `loan_status = 1`, donc la classe minoritaire. Plus tard nous allons devoir envisager une méthode pour l'équilibrer afin de mieux la prédire, sinon mon modèle aurait des tendances à ne prédire que des `loan_status = 0`.

Dans la dernière partie de l'EDA je regarde plus en détail la répartition de certaines features (voir notebook pour les graphiques) et donc on peut mieux comprendre certains liens directs entre le statut du prêt et ces dernières, par exemple beaucoup de prêts acceptés, 22% pour des locations. Enfin on regarde les distributions et les valeurs aberrantes avec des boxplots. Grâce à ces observations je peux normaliser certaines features et supprimer sans trop réduire le dataset. (étapes 2 pre-processing).

On peut voir que la distribution de certaines features sont très skewed avec de longues queues à droites et aussi beaucoup de valeurs aberrantes qui peuvent nuirent à des modèles de régression et des réseaux de neurones.

## II) Outliers, Encodage et Min\_Max Normalisation

Dans un premier temps le code supprime les valeurs aberrantes voir graphiques notebook

Encode la variable `loan_grade`, réparti de A, B, C, D, E, F, G en 1,2,3,4,5,6,7 et grâce au One-Hot-Encoding, encode le reste des variables catégoriques: `person_home_ownership`, `loan_intent`, `cb_person_default_on_file`.

On vérifie le type de toutes les nouvelles features du dataset afin de voir si elles sont toutes numériques. Voir graphique notebook

Le code applique ensuite une normalisation Min-Max aux données X pour les ramener dans un intervalle [0, 1]. Les données originales sont copiées pour permettre une comparaison. Des histogrammes sont générés pour visualiser les distributions de 5 caractéristiques avant et après normalisation, illustrant l'effet de la transformation sur les données.

## III) Pipeline, modèles, métriques AUC, entraînements

On split les données en train/ validation cela veut dire que d'une part on entraîne nos modèles sur les données d'entraînements, pour que les modèles "apprennent" et ensuite on le test via une portion du dataset de train qu'on nomme validation. Cela permet de ne pas utiliser le dataset de test et de quand même pouvoir valider, améliorer ou ajuster un modèle.

Le `pos_weight` dans PyTorch est un paramètre utilisé dans la fonction de perte `BCEWithLogitsLoss` pour gérer le déséquilibre des classes.

1. Il donne plus d'importance aux erreurs sur la classe minoritaire (positive), en multipliant la perte associée aux exemples positifs par `pos_weight`.
2. Cela aide le modèle à mieux prendre en compte les classes minoritaires dans des datasets déséquilibrés.

### Modèle de Régression Logistique:

On définit ensuite deux modèles, un modèle de régression logistique, qui permet de faire une classification binaire simple, puisqu'il s'agit d'un problème de classification binaire multiple, on peut écrire de la forme:

$$P(Y=1 | X=x_i) = \frac{e^{(a_1X_1 + a_2X_2 + \dots + a_nX_n + b)}}{1 + e^{(a_1X_1 + a_2X_2 + \dots + a_nX_n + b)}}$$

Où Y ressemble la variable `loan_status` (target) et b le bias et  $x_i$  les features, donc en d'autres termes on cherche à expliquer Y à l'aide de variables explicatives  $x_i$ .

### Configuration:

Nombre de dimensions d'entrée, une seule couche entièrement connectée reliant l'entrée (de taille `input_dim`) à une unité de sortie (1 neurone), le modèle est strictement linéaire donc pas d'autres couches cachées. L'entrée passe directement par la couche linéaire, qui calcule une combinaison linéaire pondérée des caractéristiques d'entrée. La sortie est brute c'est à dire qu'elle donne des logits mais la fonction sigmoïde permettra de transformer les logits en probabilités

Pourquoi avoir fait ce choix de configuration ?

Code simple et efficace, sortie brute et enfin pratique pour la classification.

### Points à améliorer:

J'aurais pu obtenir une meilleure précision en tâtonnant sur les métriques comme le  $R^2$  de Mac-Fadden ou en faisant des tests d'hypothèses pour mieux comprendre le pouvoir explicatif des features sur la target et ainsi sur les prédictions. Le langage R aurait pu être un bon compromis pour la partie EDA de la régression logistique mais ce n'est pas ce qui est demandé ici

## Modèle Réseaux de Neurone:

Un (MLP) avec plusieurs couches entièrement connectées, utilisé pour des tâches de prédiction.

Etapes:

1. Nombre de dimensions d'entrée (caractéristiques).
2. fc1, fc2, fc3: Couches entièrement connectées (ou Linear).
3. fc1: Connecte l'entrée à une couche cachée de 128 neurones.
4. fc2: Connecte les 128 neurones à une autre couche cachée de 64 neurones.
5. fc3: Connecte les 64 neurones à la sortie (1 seule unité pour une tâche de régression ou classification binaire).
6. relu: Fonction d'activation ReLU pour introduire de la non-linéarité contrairement à la régression logistique qui traite des données linéairement séparable et linéaire..
7. dropout: Utilisé pour éviter le sur-apprentissage en désactivant aléatoirement 30% des neurones à chaque itération. (Vu en cours de méthode d'apprentissage).
8. L'entrée passe par fc1, puis est transformée par ReLU et Dropout.
9. Le résultat est ensuite passé à fc2, à nouveau transformé par ReLU et Dropout.
10. Enfin, fc3 génère la sortie brute sans activation finale, la sortie sera passée à une fonction sigmoïde pour obtenir des probabilités plus tard

## Pourquoi avoir fait ce choix de configuration ?

En effet j'ai choisi de mettre 128 neurones et 64 neurones pour essayer de capturer les relations complexes sans trop de risque d'overfitting, avec en plus une réduction progressive de la taille des couches et avec 1 neurone en sortie ce qui est approprié pour ce type de problème, de plus la fonction d'activation ReLU est appliqué après chaque couche pour intégrer de la non-linéarité, elle est plus simple et rapide pour les calculs contrairement à sigmoid ou tanh, dans le cas de réseaux de neurones très profond, elle aide à réduire le risque de gradient vanishing ou exploding, ce qui n'est pas le cas ici. Le dropout vu en cours de Deep Learning permet de prévenir le risque d'overfitting en désactivant aléatoirement  $0.3 \Leftrightarrow 30\%$  des neurones pendant la phase de train et permet donc une meilleure généralisation. La fonction de perte est la BCE, pour

Binary Cross Entropy (stable et conforme aux attentes), optimizer Adam qui combine descente de gradient et adaptation dynamique du learning rate, la L2 régularisation aide aussi à la généralisation.

### Métriques

La fonction accuracy calcule l'exactitude du modèle en transformant les sorties brutes en probabilités via une sigmoïde, puis en prédictions binaires basées sur un seuil (0.5). Elle retourne le pourcentage de prédictions correctes (TP + TN / Total).

Ensuite la fonction compute\_roc\_auc, trouvée en ligne, calcule l'AUC (aire sous la courbe ROC). Elle le fait en triant les scores prédits, en calculant les taux de vrais positifs (TPR) et de faux positifs (FPR) à différents seuils, et en intégrant la courbe pour obtenir l'AUC. Cette métrique, purement informative, mesure la capacité du modèle à discriminer entre les classes.

### Entraînement:

La fonction train\_model entraîne un modèle sur des données tabulaires avec PyTorch. Elle utilise une loss BCEWithLogitsLoss, adaptée aux tâches de classification binaire, et intègre une régularisation L2 pour éviter le surapprentissage. Les données sont converties en tenseurs PyTorch, et l'entraînement est effectué par batch\_size c'est-à-dire mini-lots. À chaque epoch :

1. La perte, l'accuracy, et l'AUC sont calculées.
2. Les prédictions sont évaluées sur l'ensemble d'entraînement.
3. La meilleure AUC est mise à jour si elle s'améliore.

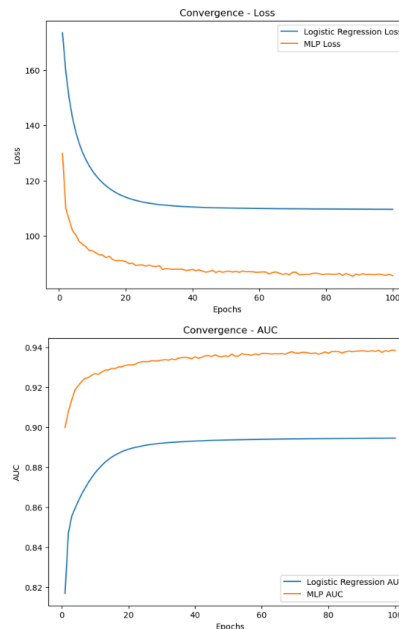
## IV) Comparaisons et figures de convergences

Les métriques à chaque epoch sont stockées et imprimées pour analyse post-entraînement. La fonction retourne ces métriques et la meilleure AUC obtenue.

Le code exécute et compare ensuite les deux modèles sur l'ensemble de train, j'avais effectué à la base sur l'ensemble de validation mais je l'ai retiré du code pour plus de clarté:

```
=== Entraînement du modèle MLP (rééquilibré) ===
Epoch 1/100, Train Loss: 129.7486, Train Acc: 82.58%, Train AUC: 0.8998
Epoch 2/100, Train Loss: 109.8182, Train Acc: 83.42%, Train AUC: 0.9079
Epoch 3/100, Train Loss: 105.7631, Train Acc: 83.84%, Train AUC: 0.9136
Epoch 4/100, Train Loss: 101.8105, Train Acc: 84.33%, Train AUC: 0.9187
Epoch 5/100, Train Loss: 100.2121, Train Acc: 84.31%, Train AUC: 0.9208
Epoch 6/100, Train Loss: 97.9622, Train Acc: 84.73%, Train AUC: 0.9228
Epoch 7/100, Train Loss: 96.9186, Train Acc: 84.84%, Train AUC: 0.9243
Epoch 8/100, Train Loss: 96.0073, Train Acc: 84.73%, Train AUC: 0.9249
Epoch 9/100, Train Loss: 94.6862, Train Acc: 85.34%, Train AUC: 0.9260
Epoch 10/100, Train Loss: 94.4708, Train Acc: 85.20%, Train AUC: 0.9268
Epoch 11/100, Train Loss: 93.7701, Train Acc: 84.92%, Train AUC: 0.9264
Epoch 12/100, Train Loss: 93.0598, Train Acc: 85.26%, Train AUC: 0.9275
Epoch 13/100, Train Loss: 92.9847, Train Acc: 85.65%, Train AUC: 0.9285
Epoch 14/100, Train Loss: 92.0508, Train Acc: 85.33%, Train AUC: 0.9286
Epoch 15/100, Train Loss: 92.5361, Train Acc: 85.82%, Train AUC: 0.9293
Epoch 16/100, Train Loss: 91.3738, Train Acc: 85.56%, Train AUC: 0.9293
Epoch 17/100, Train Loss: 90.9019, Train Acc: 85.90%, Train AUC: 0.9301
Epoch 18/100, Train Loss: 90.9930, Train Acc: 85.85%, Train AUC: 0.9302
Epoch 19/100, Train Loss: 90.8656, Train Acc: 85.99%, Train AUC: 0.9309
Epoch 20/100, Train Loss: 90.6157, Train Acc: 85.95%, Train AUC: 0.9311
Epoch 21/100, Train Loss: 89.8334, Train Acc: 86.14%, Train AUC: 0.9311
Epoch 22/100, Train Loss: 90.0066, Train Acc: 85.84%, Train AUC: 0.9314
Epoch 23/100, Train Loss: 89.1955, Train Acc: 86.20%, Train AUC: 0.9323
Epoch 24/100, Train Loss: 89.3762, Train Acc: 86.22%, Train AUC: 0.9327
Epoch 25/100, Train Loss: 89.3789, Train Acc: 86.23%, Train AUC: 0.9327
Epoch 26/100, Train Loss: 88.9510, Train Acc: 86.23%, Train AUC: 0.9328
Epoch 27/100, Train Loss: 89.3522, Train Acc: 86.31%, Train AUC: 0.9332
Epoch 28/100, Train Loss: 88.9370, Train Acc: 86.27%, Train AUC: 0.9331
Epoch 29/100, Train Loss: 88.7308, Train Acc: 86.18%, Train AUC: 0.9332
Epoch 30/100, Train Loss: 89.1319, Train Acc: 86.34%, Train AUC: 0.9335
Epoch 31/100, Train Loss: 87.7443, Train Acc: 86.38%, Train AUC: 0.9338
```

On observe ici pour le MLP que la loss diminue à mesure que les époques augmentent, donc la descente de gradient est efficace, on voit aussi que l'AUC du modèle augmente aussi dans ce sens. Même chose que pour la régression logistique mais avec une précision différente.





Le graphique de convergence de la loss (perte) et de l'AUC sont pertinents car on voit bien à partir de quelle époque la loss cesse de diminuer et aussi les différences de performance entre les deux modèles ex: à partir de 45/50 époques la loss ne diminue quasiment plus et la courbe ROC vers 50 stagne aussi, donc c'est un bon indicateur de performance surtout sur de gros modèle de deep learning, il faut absolument regarder comment la loss converge en temps réels pour ne pas avoir de mauvaises surprises ex: weight&biases le permet.(VAE en cours de deep learning pour la génération d'images).

Enfin avant de créer le test set, je tente de chercher une optimisation de ces hyper-parametres. Je teste ici différents taux d'apprentissage pour le modèle MLP. À chaque taux (lr\_test), le modèle est entraîné, et l'AUC obtenue est comparée pour identifier le meilleur taux d'apprentissage. Le best\_lr est sélectionné comme celui ayant produit la meilleure AUC après 50 epochs.


Il faudrait que je teste une plage plus large ou utiliser des techniques comme la recherche aléatoire ou bayésienne pour explorer les hyperparamètres  
Je pourrais aussi tester sur d'autres hyper-paramètres pour le L2, le dropout etc mais ça dépasse mon champ de compétences.

## V) Soumission Kaggle et NEF

Pour finir avant de soumettre je génère un fichier de test.csv contenant toutes mes prédictions via le modèle MLP sur la target.

Le fichier est de la même forme que le submission\_sample et respecte les mêmes caractéristiques et pre-processing que le train.csv.

Je peux donc le fournir à Kaggle et obtient:

Submission and Description		Private Score ⓘ	Public Score ⓘ	Selected
	<b>submission.csv</b> Complete (after deadline) · 5h ago	<b>0.85754</b>	<b>0.86496</b>	<input type="checkbox"/>

<https://www.kaggle.com/mariuscasamian>

Score correct mais qui peut être encore mieux si j'utilisais des modules comme scikit-learn, utilisait des méthodes d'ensemble learning (boosting, stacking etc)

ou alors en tentant de rechercher les meilleurs paramètres pour les modèles actuels de MLP et régression car ils sont mal ou pas bien optimisés.

Pour le NEF j'ai eu un message d'erreur suivant:

```
[~/Bureau]
❌ base m1000 ssh mcasamia@nef-frontal.inria.fr
Your account has expired; please contact your system administrator
Connection closed by 193.51.209.228 port 22
```

C'est pourquoi je n'ai pas pu le tester, mais puisque je l'ai utilisé pendant mon semestre et sait comment m'en servir, voici les commandes que j'aurais tapées:

- 1) Ouvrir mon nef portal en me connectant dessus via  
> ssh [mcasamia@nef-frontal.inria.fr](mailto:mcasamia@nef-frontal.inria.fr)
- 2) > notebook-nbconvert -to script  
Marius\_Casamian\_python\_projet\_ia\_ipynb pour extraire le code python du notebook
- 3) Nettoyer proprement le fichier python pour ne garder que les modèles
- 4) Copier le dossier contenant les fichiers csv et le script python des modèles vers le NEF  
> scp -r mon\_fichier mcasamia@nef-frontal:~
- 5) Installer conda et les dépendances  
> conda create --name Mon\_Projet  
> conda activate Mon\_Projet  
> cd mon\_fichier  
> conda install requirements.txt
- 6) Se connecter au ssh devel  
> ssh nef-devel
- 7) Réserver un node  
> oarsub -p "gpu='YES'" -l /gpunum=2,walltime=4:0:0 -I  
YES pour spécifier les nodes de GPU disponibles
- 8) Si la connexion au node ne s'est pas faite directement alors taper  
> oarsub -C id
- 9) Enfin exécuter le main.py
- 10) Terminer et conclure avant de kill le pid du node

Fin du compte rendu, projet intéressant qui me donne une idée de ce genre de compétition et qui me pousse à apprendre encore pour faire mieux à la prochaine.