🧠 Analyse des goulots d'étranglement via le profiling

Dans le cadre de l'optimisation des performances de l'API, j'ai mis en place un **profiling ciblé** de la fonction clé `predict_credit_score`, responsable du calcul des scores de crédit.
L'objectif était d'**identifier les éventuels goulots d'étranglement** afin de guider les optimisations.

🛠️ Méthodologie :

1. **Profiling avec `cProfile`** :
   J'ai encapsulé la fonction `predict_credit_score` dans un script de profiling Python à l'aide du module `cProfile`, pour mesurer les temps d'exécution de chaque fonction appelée.

2. **Visualisation avec `SnakeViz`** :
   Les résultats du profiling ont été enregistrés dans un fichier `.prof` et visualisés avec **SnakeViz**, un outil interactif permettant de repérer visuellement les fonctions les plus consommatrices.

3. **Analyse des temps cumulatifs (`cumtime`)** :
   En me basant sur la largeur des blocs supérieurs dans le diagramme (temps d'exécution total d'une fonction et de ses appels enfants), j'ai identifié les zones critiques.

🔍 Goulots d'étranglement identifiés :

- **`data.py:566 (oth_type)` (~0.055 s)**
  Cette fonction interne à Pandas est liée à la **gestion ou conversion des types de données**, indiquant un coût élevé dans la préparation des données avant prédiction.

- **`base.py:541 (to_numpy)` (~0.052 s)**
  Cette opération est appelée lors de la conversion du DataFrame vers un tableau NumPy, suggérant que **les transformations de données** représentent une part significative du temps d'inférence.

✅ Conclusion :

Le profiling met en évidence que **le principal goulot d'étranglement ne se situe pas dans le modèle lui-même**, mais dans la **préparation des données** (notamment les conversions Pandas → NumPy).
Ces opérations pourraient être optimisées ou évitées, par exemple en standardisant les entrées dès la réception de la requête, ou en utilisant des structures plus légères comme NumPy natif si possible.

🔧 Optimisation de la fonction d'inférence dans mon API Gradio

Dans le cadre de mon projet, j'ai intégré une fonction d'inférence `predict_credit_score` dans une API Gradio, permettant de prédire le risque de crédit d'un client à partir de 10 variables simplifiées. Initialement, cette fonction prenait les 10 variables en entrée, puis construisait un dictionnaire complet contenant **l'ensemble des features du modèle** (plusieurs centaines), en initialisant toutes les autres variables à zéro. Ensuite, ce dictionnaire était converti en un `DataFrame` Pandas, en respectant l'ordre des colonnes attendu par le modèle.

Cette méthode, bien que fonctionnelle, introduisait une **latence inutile**. Le passage par Pandas est coûteux en temps pour un seul échantillon, car il implique de nombreuses opérations de conversion de types, de gestion des valeurs manquantes (`NaN`), et d'allocation mémoire non négligeable. Grâce à l'outil de profilage **SnakeViz**, j'ai pu visualiser précisément les étapes les plus chronophages. Les blocs dominants dans le graphique étaient liés à des fonctions internes de Pandas, comme `data.py`, `missing.py` et `series.py`, ce qui confirmait que la majeure partie du temps était consacrée à la **préparation des données**, et non à l'inférence du modèle lui-même.

Pour améliorer cela, j'ai remplacé la préparation des données via Pandas par une construction **directe d'un tableau NumPy**, compatible avec XGBoost. Techniquement, j'ai créé un tableau `np.zeros((1, len(all_features)), dtype=np.float32)`, c'est-à-dire une ligne vide avec toutes les colonnes attendues par le modèle. Ensuite, j'ai simplement inséré les 10 valeurs saisies par l'utilisateur dans les bonnes positions (repérées grâce aux index dans `all_features`). Cela permet de **bypasser totalement Pandas**, tout en assurant que le modèle reçoit exactement le même format qu'à l'entraînement.

Après cette optimisation, j'ai de nouveau profilé la fonction avec SnakeViz. Le graphique obtenu montre une structure radicalement différente : le temps est désormais **entièrement consacré à la fonction `predict_proba` de XGBoost**, ce qui signifie que le goulot d'étranglement lié à la préparation des données a été éliminé. Le temps total d'inférence est passé d'environ **50-60 millisecondes à 1,6 milliseconde**, ce qui représente un gain considérable pour une application temps réel.

Pour observer les différences dans snakeviz, lancer d'abord snakeviz avec profiling_output.prof (avant opti) et ensuite lancer snakeviz avec profiling_output3.prof (après opti du code).

Cette démarche m'a permis d'identifier et de corriger un goulot d'étranglement non lié au modèle, mais à l'environnement de préparation des données. C'est un point crucial dans toute API de machine learning : **le modèle peut être rapide, mais mal préparé, il devient lent.** Le bon usage des outils de profilage comme `cProfile` et `SnakeViz` m'a donc permis de prioriser mes optimisations et de prouver leur efficacité de manière mesurable et visuelle.

🔍 Avant l'optimisation du code `predict_credit_score` :

Prenons les trois premières entrées de log (avant le 8 septembre) :

- Durée moyenne d'inférence : ~ 0.0457 à 0.1027 secondes
- CPU : entre 0.0 % et 18.7 %
- Mémoire : autour de 260–263 Mo

👉 L'inférence restait relativement rapide, mais on observe des durées non négligeables (jusqu'à 0.1 seconde), en particulier pour de petites entrées. Cela peut s'expliquer par des étapes inutiles ou répétées dans la construction du `DataFrame`, notamment la réinitialisation complète de toutes les features à 0, puis leur mise à jour à chaque appel.

✅ Après l'optimisation (logs du 8 septembre) :

- Durée moyenne d'inférence : 0.001 à 0.0046 secondes
- CPU : stable, jusqu'à 18.6 %
- Mémoire : légèrement réduite (~231–260 Mo)

👉 Les résultats sont clairement plus performants :

- La latence est divisée par 10 à 50 : l'inférence est maintenant quasi-instantanée (environ 1 milliseconde).
- L'utilisation mémoire est légèrement optimisée (probablement due à une meilleure gestion du `DataFrame` temporaire).
- La performance CPU reste cohérente (pas de surchauffe constatée).

Essai de ONNX (échec)

Dans une logique d'optimisation et de portabilité du modèle, j'ai souhaité tester la conversion du modèle XGBoost au format ONNX (Open Neural Network Exchange). L'objectif était de bénéficier des avantages que propose ONNX en production. Pour cela, j'ai utilisé la librairie `onnxmltools` afin de convertir le modèle `.pkl` en format `.onnx`. Une fois le modèle converti, j'ai intégré `onnxruntime.InferenceSession` dans mon pipeline pour mesurer précisément les performances.

📝 Comparaison des performances d’inférence : XGBoost vs ONNX

Dans le cadre de l’optimisation du modèle pour un déploiement en production, nous avons comparé les temps d’inférence du modèle original XGBoost (au format .pkl) avec sa version convertie en ONNX.

⏱️ Résultats observés :

Modèle XGBoost (.pkl) : 0.062 secondes

Modèle ONNX (via onnxruntime) : 0.119 secondes

⚖️ Analyse :

Le modèle original XGBoost s’avère presque deux fois plus rapide que sa version ONNX pour un batch de 48 744 échantillons. Cette différence s'explique principalement par :

XGBoost est déjà hautement optimisé pour l'inférence CPU. Son implémentation en C++ offre un accès direct à la mémoire et une exécution très rapide.

Le modèle ONNX, bien que portable, nécessite une initialisation de session et un encodage spécifique des entrées, ce qui introduit une latence supplémentaire.

Dans notre cas, le modèle est relativement léger (nombre limité de features, faible profondeur), ce qui limite les bénéfices d’optimisation offerts par ONNX.

✅ Conclusion :

Le modèle XGBoost d'origine est plus performant en termes de vitesse d'inférence dans notre contexte, sans perte de précision. Le choix du modèle .pkl est donc plus adapté pour une utilisation en production sur CPU standard, à moins qu’une portabilité multi-environnements ne soit requise.