
Make Us Rich

Outil de prédiction de l'évolution des cours des
crypto-monnaies

Thomas Chaigneau

May 14, 2022

Contents

1	Introduction	5
1.1	Contexte	5
1.2	Enoncé du problème	5
1.3	Objectifs	5
1.4	Approche de la solution	5
1.5	Contributions et réalisations	6
1.6	Organisation du rapport	7
2	Etat de l'art	8
2.1	Aperçu et historique rapide	8
2.2	Fondamentaux des séries temporelles	9
2.3	Différents modèles de prédiction	10
2.4	Réseaux de neurones récurrents (<i>RNN</i>)	12
2.4.1	Architecture du réseau de neurones récurrents	12
2.4.2	Avantages et inconvénients	13
2.4.3	Gestion des gradients	14
2.4.3.1	Cas de gradient qui explose	14
2.4.3.2	Cas de gradient qui disparaît	15
2.4.4	Fonctions d'activation	16
2.4.5	GRU et LSTM	16
2.4.5.1	Gated Recurrent Unit (GRU)	17
2.4.5.2	Long Short-Term Memory (LSTM)	17
2.5	L'avènement des modèles Transformers	18
3	Make Us Rich	20
3.1	Les données	20
3.1.1	Récupération des données	20
3.1.2	Description des données	21
3.1.3	Préparation des données	22
3.1.3.1	Extraction des données utiles	22
3.1.3.2	Séparation des jeux de données	22
3.1.3.3	Mise à l'échelle des données	22
3.1.3.4	Préparation des séquences de données	23
3.2	Modélisation	23
3.2.1	Définition de l'architecture du modèle	24
3.2.2	Choix des hyperparamètres	24

3.2.3	Entraînements et monitoring	25
3.2.4	Validation du modèle	25
3.2.5	Conversion vers ONNX	26
3.2.6	Stockage des modèles et des <i>features engineering</i>	26
3.3	Service des modèles	27
3.3.1	Qu'est-ce-que servir des modèles	27
3.3.2	Dockerisation	27
3.3.3	Présentation de l'API	28
	3.3.3.1 Gestion des modèles	28
	3.3.3.2 Les endpoints de l'API	28
3.4	Interface utilisateur	29
3.5	Packaging du projet	29
4	Conclusion	30

Je, Thomas Chaigneau, de l'école Microsoft IA by Simplon à Brest, en alternance en tant que Développeur IA au Crédit Mutuel Arkéa, confirme que c'est mon propre travail et les figures, les tableaux, les extraits de code et les illustrations dans ce rapport sont originaux et n'ont pas été tirés de l'œuvre d'aucune autre personne, sauf lorsque les œuvres d'autres ont été explicitement reconnues, citées et référencées. Je comprends que cela sera considéré comme un cas de plagiat, sinon. Le plagiat est une forme d'inconduite académique et sera pénalisé en conséquence.

Je donne mon consentement à ce qu'une copie de mon rapport soit communiquée aux futurs étudiants comme exemple.

Je donne mon consentement pour que mon travail soit rendu plus largement accessible au public avec un intérêt pour l'enseignement, l'apprentissage et la recherche.

Thomas Chaigneau, April 22, 2022

Tuteur : *Jean-Marie Prigent, Machine Learning Engineer @ Crédit Mutuel Arkéa*

Un rapport soumis en réponse aux exigences de la formation *Développeur IA* de l'école *Microsoft IA by Simplon à Brest* pour le titre RNCP34757.

1 Introduction

1.1 Contexte

Aujourd'hui, il existe plusieurs solutions de surveillance de portefeuilles financiers en ligne, qui permettent le suivi de l'évolution du cours de différents actifs. Dans le cas des crypto-monnaies, ces outils offrent à l'utilisateur une interface gratuite et simple lui permettant d'ajouter manuellement ses actifs et de les suivre en temps réel. Aucun, à ma connaissance, n'intègre des outils d'analyses et de prédictions des cours des crypto-monnaies.

1.2 Enoncé du problème

Puisqu'aucun outil, disponible gratuitement, n'offre la possibilité de simuler une prédiction de l'évolution des cours des crypto-monnaies, je souhaite créer un outil permettant de façon automatisée et transparente pour l'utilisateur de suivre l'évolution des cours des crypto-monnaies. Comment permettre à l'utilisateur de disposer d'informations de prédictions automatiques sur ses actifs qui soient un minimum fiables ?

1.3 Objectifs

Développer une architecture permettant l'entraînement automatique de modèles de prédiction de l'évolution des cours des crypto-monnaies, le service de ces modèles via une *API REST* ainsi qu'une interface web permettant de visualiser les prédictions et l'évolution des cours des crypto-monnaies par l'utilisateur.

1.4 Approche de la solution

Le projet se décompose en trois composants distincts et qui interagissent entre eux pour former la solution complète :

- **Interface** : l'interface web permettant à l'utilisateur de visualiser les prédictions et l'évolution des cours des crypto-monnaies.
- **Serving** : le serveur web qui met à disposition les modèles de prédiction via une *API REST*.
- **Training** : pipeline automatisé d'entraînement des modèles, de leur validation et de leur stockage.

Voici un schéma de l'architecture du projet :

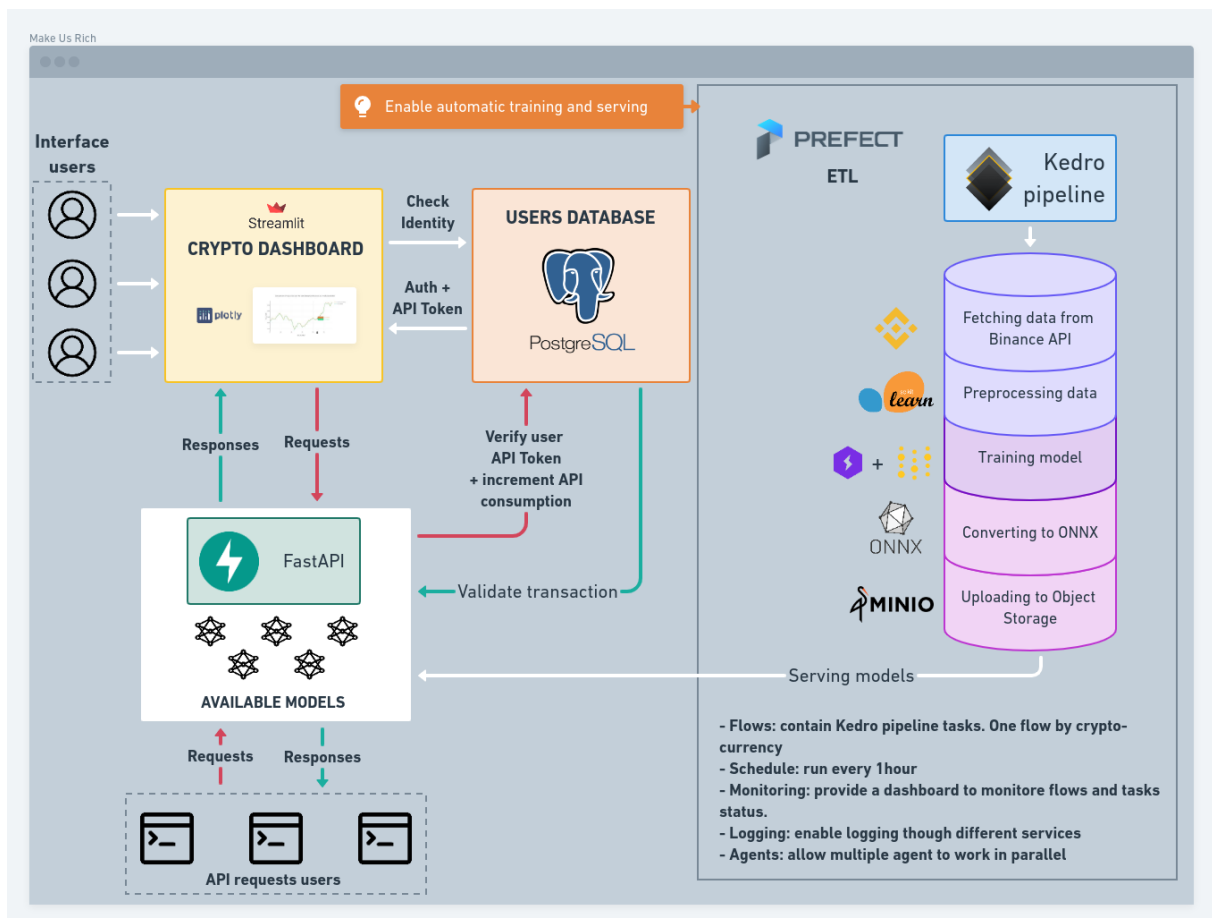


Figure 1: Architecture du projet Make Us Rich

Ce schéma très complet reprend tous les composants du projet et présente leurs interactions. Nous détaillerons les différents composants et leurs fonctions dans la seconde partie de ce rapport.

1.5 Contributions et réalisations

Tout ce qui est présenté dans le schéma d'architecture ci-dessus est fonctionnel et déployé. Les composants sont développés en utilisant le langage de programmation Python et différents outils et librairies très utiles comme *FastAPI*, *Pytorch-Lightning*, *Scikit-learn*, etc.

Le projet est open-source et est accessible sur *GitHub* : Make Us Rich.

Il est possible d'y contribuer et de l'améliorer. Il est également possible de simplement l'utiliser

et de déployer localement tous les composants du projet. Toutes les étapes de déploiement sont détaillées dans la documentation associée qui est également disponible sur *GitHub* : Documentation.

1.6 Organisation du rapport

Le rapport s'organise en 3 grandes parties :

- **Etat de l'art** : présentation des avancées des modélisations IA et des algorithmes de prédiction sur des séries temporelles.
- **Make Us Rich** : présentation et détails de la solution et de son architecture.
- **Conclusion** : retour sur le projet et ouverture sur les différents axes d'amélioration du projet.

2 Etat de l'art

Dans cette première section, nous allons décrire les avancées de la modélisation IA et des algorithmes de prédiction sur des données temporelles. Ce ne sera malheureusement pas une liste exhaustive de toutes les options disponibles, ni un historique complet des différentes évolutions des algorithmes de prédiction sur des séries temporelles, car cela est trop riche pour tenir dans ce rapport.

Nous allons donc nous concentrer sur les avancées les plus récentes et celles qui ont un rapport direct avec la solution envisagée dans ce projet. Commençons par un bref aperçu et historique de la tâche de prédiction à l'aide de données temporelles.

2.1 Aperçu et historique rapide

Les séries temporelles, ainsi que leur analyse, sont de plus en plus importantes en raison de la production massive de données dans le monde. Il y a donc un besoin, en constante augmentation, dans l'analyse de ces séries chronologiques avec des techniques statistiques et plus récemment d'apprentissage automatique.

L'analyse des séries chronologiques consiste à extraire des informations récapitulatives et statistiques significatives à partir de points classés par ordre chronologique. L'intérêt étant de diagnostiquer le comportement passé pour prédire le comportement futur.

Aucune des techniques ne s'est développée dans le vide ou par intérêt purement théorique. Les innovations dans l'analyse des séries chronologiques résultent de nouvelles méthodes de collecte, d'enregistrement et de visualisation des données (Nielsen 2019).

Il existe énormément de domaines d'application tels que la médecine, la météorologie, l'astronomie ou encore ce qui va nous intéresser ici, les marchés financiers et notamment celui des crypto-monnaies.

Pour revenir à l'aspect historique, les organisations privées et notamment bancaires ont commencé à collecter des données par imitation du gouvernement américain qui collectait des données économiques publiques. Les premiers pionniers de l'analyse des données chronologiques des cours de la bourse ont fait ce travail mathématique à la main, alors que de nos jours ce travail est réalisé avec l'assistance de méthodes analytiques et des algorithmes de machine learning (Nielsen 2019).

Richard Dennis, dans les années 80, a été le premier à développer un algorithme de prédiction des cours de la bourse qui ne comprenait que quelques règles de base, permettant à quiconque les connaissant de prévoir le prix d'une action et d'en retirer des bénéfices par spéculation.

Progressivement, avec l'accumulation de personnes utilisant ces règles, elles sont devenues de plus en plus inefficaces. Il aura donc fallu développer de nouvelles méthodes, notamment statistiques, plus complexes pour toujours mieux prévoir l'évolution des cours des marchés financiers.

C'est ainsi que des méthodes historiques telles que *SARIMA* ou *ARIMA* se sont démocratisées. Elles présentent néanmoins un inconvénient : elles nécessitent des données stationnaires pour fonctionner. De plus, ces techniques statistiques dites historiques ont des résultats médiocres sur le long terme, et ainsi se sont développés d'autres méthodes d'apprentissage automatique utilisant la puissance des réseaux de neurones, comme *RNN* (Recurrent Neural Network) (Rumelhart and Williams 1985).

2.2 Fondamentaux des séries temporelles

Comme évoqué précédemment, la stationnarité d'une série est une propriété essentielle pour l'analyse statistique. Une série chronologique est dite stationnaire si ces propriétés telles que la moyenne, la variance ou la covariance sont constantes au cours du temps. Or, cela n'est pas vrai pour toutes les séries temporelles et notamment les données issues des marchés financiers, qui ne sont stationnaires que sur une période de temps fixée (souvent courte).

Il existe trois composantes qui constituent une série temporelle :

- **Tendance (T = Trend)** : correspond à une augmentation ou à une diminution sur le long terme des données et qui peut assumer une grande variété de modèles. Nous utilisons la tendance pour estimer le niveau, c'est-à-dire la valeur ou la plage typique de valeurs, que la variable doit avoir au cours du temps. On parle de tendance à la hausse ou à la baisse.
- **Saisonnalité (S = Seasonal)** : est l'apparition de schémas de variations cycliques qui se répètent à des taux de fréquence relativement constants.
- **Résidu (R = Remainder)** : correspondent aux fluctuations à court terme qui ne sont ni systématiques ni prévisibles. Au quotidien, des événements imprévus provoquent de telles instabilités. Concrètement, la composante résiduelle est ce qui reste après l'estimation de la tendance et de la saisonnalité, et leur suppression d'une série chronologique.

Voici une représentation de la décomposition des composantes d'une série temporelle :

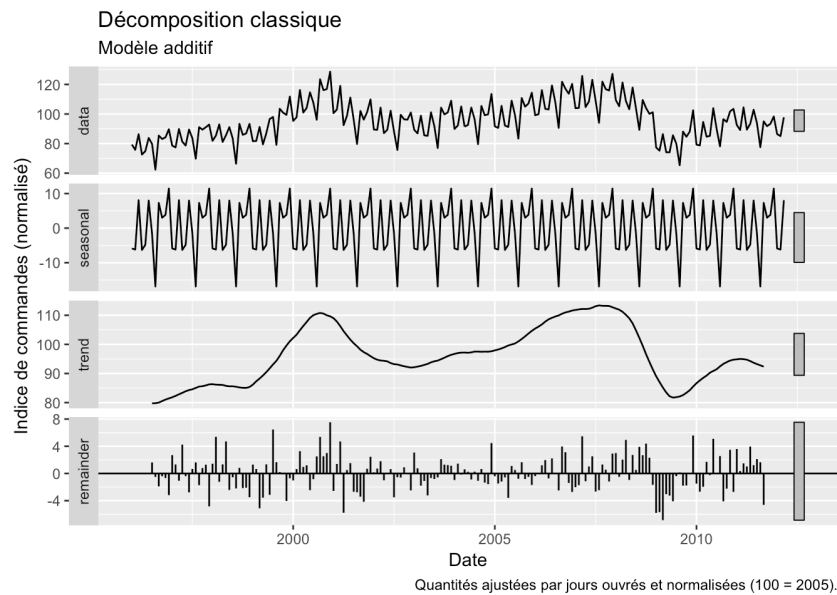


Figure 2: Graphique présentant la décomposition classique d'un modèle additif d'une série temporelle (Chailan and Palacios-Rodríguez 2018)

2.3 Différents modèles de prédiction

Nous allons voir ici les différentes techniques et modèles qui existent pour la prédiction à l'aide de données temporelles.

Nous pouvons d'ores et déjà distinguer deux catégories de modèles de prédiction :

- **Modèles statistiques** : dits traditionnels, ils regroupent les modèles univariés et multivariés comprenant respectivement *ARIMA*, *SARIMA* et *VAR*.

Un processus stationnaire X_t admet une représentation $ARIMA(p, d, q)$ dite minimale s'il existe une relation (Goude 2020) :

$$\Phi(L)(1 - L)^d X_t = \Theta(L)\epsilon_t, \forall_t \in Z$$

avec pour conditions :

- $\phi_p \neq 0$ et $\theta_q \neq 0$
- Φ et Θ doivent être des polynômes de degrés respectifs p et q , n'ont pas de racines communes et leurs racines sont de modules > 1
- ϵ_t est un BB de variance σ^2

De même, un processus stationnaire X_t admet une représentation $SARIMA(p, d, q)$ dite minimale si la relation suivante est vraie (Goude 2020) :

$$(1 - L)^d \Phi_p(L) (1 - L^s)^D \Phi_P(L^s) X_t = \theta_q(L) \theta_Q(L^s) \epsilon_t, \forall_t \in Z$$

avec les mêmes conditions que pour les modèles ARIMA.

- **Modèles d'apprentissage automatique** : ils regroupent les modèles de régression par amplification de gradient et les modèles par apprentissage profond comprenant les réseaux de neurones récurrents et convolutionnels.

Voici une représentation des différents modèles de prédiction :

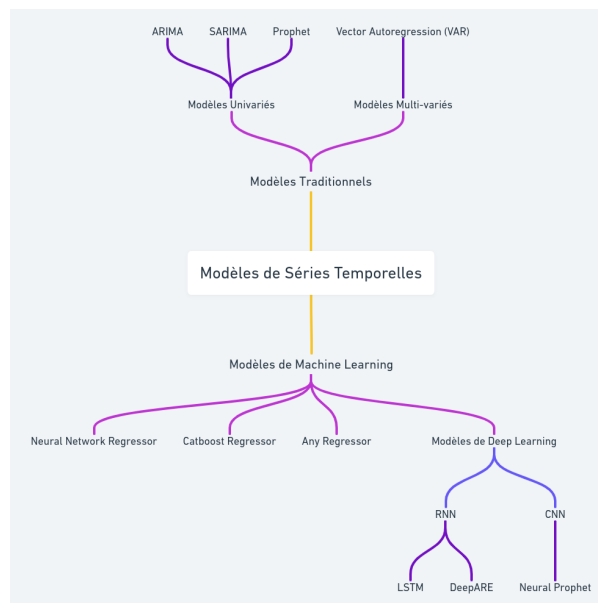


Figure 3: Liste non-exhaustive des modèles utilisés pour la prédiction de séries chronologiques.

Nous pourrions également ajouter à cette liste les très récents modèles basés sur l'architecture *Transformers*, comme **Temporal Fusion Transformers** (TFT) qui est un modèle de Google (Lim et al. 2019) qui permet de combiner des données temporelles avec des données non temporelles, des données statiques comme des informations de localisation dans le cas de prédictions météorologiques (Kafritsas 2021).

Dans notre projet, nous allons nous concentrer sur les modèles de Machine Learning les plus récents, qui sont les modèles de Deep Learning tels que les architectures réseaux de neurones convolutionnels et réseaux de neurones récurrents.

2.4 Réseaux de neurones récurrents (RNN)

Les réseaux de neurones récurrents, ou *Recurrent Neural Network*, sont des architectures de neurones qui sont utilisés dans beaucoup de cas d'usage. Ils sont appelés réseaux de neurones récurrents car ils sont capables de se réguler en fonction de la sortie des neurones précédents (Sherstinsky 2020). Ils sont notamment utilisés pour la prédiction de séries temporelles, car ils permettent de prédire la valeur d'une variable à partir de ses valeurs précédentes.

2.4.1 Architecture du réseau de neurones récurrents

C'est par le biais d'états cachés (en anglais *hidden states*) qu'un modèle RNN est capable de réaliser la prédiction d'une variable. Ainsi, un modèle RNN prend en entrée des séquences de vecteurs de données, et non pas des vecteurs de données individuels.

Une architecture traditionnelle d'un RNN se présente comme suit (Amidi and Amidi 2020) :

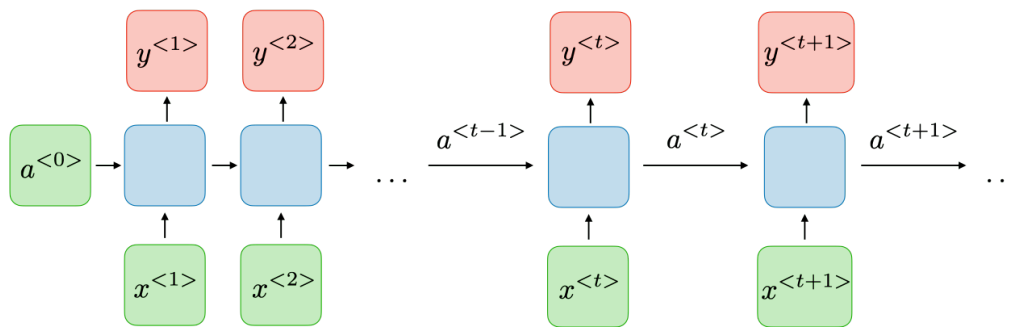


Figure 4: Architecture d'un réseau de neurones récurrents (RNN)

À l'instant t , l'activation $a^{<t>}$ d'un neurone est définie par la fonction d'activation suivante :

$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$$

et la sortie $y^{<t>}$ est de la forme :

$$y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$$

où W_{ax} , W_{aa} , W_{ya} , b_a et b_y sont des coefficients de poids partagés temporellement entre les fonctions d'activation g_1 et g_2 .

Il est également intéressant de s'intéresser à l'architecture d'une cellule (en anglais *block*) qui compose un réseau de neurones récurrents. Cela permet de comprendre les mécanismes qui

ont lieu à chaque étape de la propagation de données dans un réseau RNN. Ce sera également utile dans un second temps pour pouvoir comparer les différences majeures avec des architectures plus intéressantes que celles dites classiques, que nous verrons juste après.

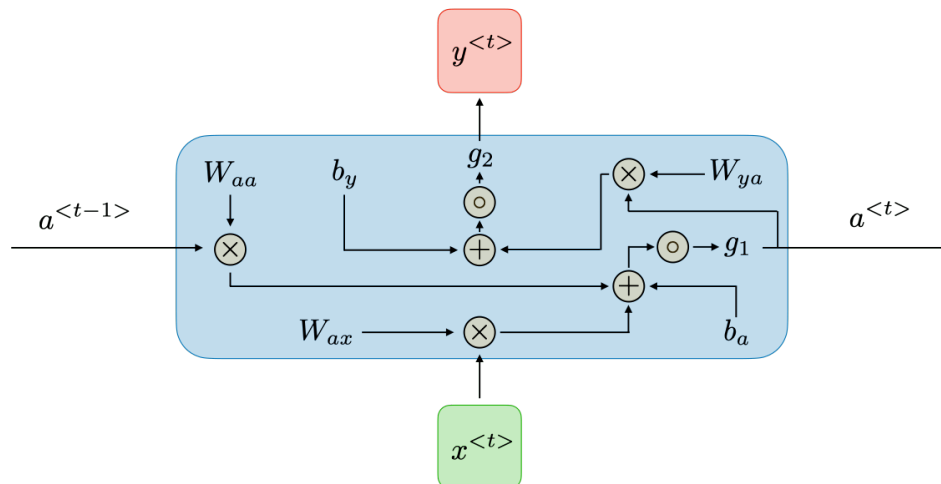


Figure 5: Architecture d'une cellule d'un réseau de neurones récurrents (RNN) (Amidi and Amidi 2020)

Nous pouvons voir que chaque cellule va prendre un état de la donnée précédente, et qu'elle va produire une sortie en fonction de son état et de la fonction d'activation associée.

2.4.2 Avantages et inconvénients

Voici un tableau récapitulatif des avantages et inconvénients d'utiliser ce genre de modélisation (Amidi and Amidi 2020) :

Table 1: Avantages et inconvénients des modèles RNN

Avantages	Inconvénients
- Possibilité de traiter une entrée de n'importe quelle longueur	- Le calcul est plus consommateur en ressources (par rapport à d'autres modèles)
- La taille du modèle n'augmente pas avec la taille de l'entrée	- Difficulté pour accéder aux informations trop lointaines
- Le calcul prend en compte les informations historiques	- Impossibilité d'envisager une entrée future pour l'état actuel

Avantages

Inconvénients

- Les pondérations sont réparties dans le temps

Nous pouvons constater que comme attendu lors de l'utilisation de modèles de *Deep Learning*, les modèles RNN sont plus consommateurs en ressources que d'autres modèles de *Machine Learning*. Ils présentent néanmoins des avantages non négligeables, en dehors d'un gain de performances, pour notre cas d'usage dans le cadre de la prédiction de séries temporelles financières.

2.4.3 Gestion des gradients

Il existe également un autre inconvénient des modèles RNN qui sont les phénomènes de gradients qui disparaissent et qui explosent lors de l'apprentissage. En anglais, on parle de *vanishing gradient* et *exploding gradient*.

Cela est expliqué par le fait que sur le long terme il est très difficile de capturer les dépendances à cause du gradient multiplicatif qui peut soit décroître, soit augmenter de manière exponentielle en fonction du nombre de couches du modèle.

2.4.3.1 Cas de gradient qui explose

Pour contrer les phénomènes de gradient qui explose, il est possible d'utiliser une technique de *gradient clipping* (Sherstinsky 2020) (en français *coupure de gradient*) qui permet de limiter le gradient à une valeur fixée. Puisque la valeur du gradient est plafonnée les phénomènes néfastes de gradient sont donc maîtrisés en pratique.

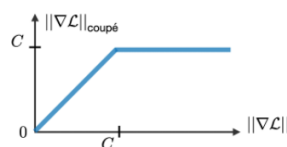


Figure 6: Technique de gradient clipping

Grâce à cette technique, nous pouvons donc éviter que le gradient devienne trop important en le remettant à une échelle plus petite.

2.4.3.2 Cas de gradient qui disparaît

Concernant les phénomènes de gradient qui disparaissent, il est possible d'utiliser des *portes* de différents types, souvent notées Γ et sont définies par :

$$\Gamma = \sigma(Wx^{<t>} + Ua^{<t-1>} + b)$$

où W , U et b sont des coefficients spécifiques à la porte et σ est une fonction sigmoïde.

Les portes sont utilisées dans les architectures plus spécifiques comme *GRU* et *LSTM* que nous verrons juste après.

Table 2: Comparaison des différents types de portes et leurs rôles

Type de porte	Rôle	Utilité
Porte d'actualisation Γ_u	Décide si l'état de la cellule doit être mis à jour avec la valeur d'activation en cours	GRU, LSTM
Porte de pertinence Γ_r	Décide si l'état de la cellule antérieure est important ou non	GRU, LSTM
Porte d'oubli Γ_f	Contrôle la quantité d'information qui est conservé ou oublié de la cellule antérieure	LSTM
Porte de sortie Γ_o	Détermine le prochain état caché en contrôlant quelle quantité d'information est libérée par la cellule	LSTM

Ces différents types de portes permettent de corriger les erreurs de calcul du gradient en fonction de la mesure de l'importance du passé, et ainsi de s'affranchir en partie des phénomènes de gradient qui disparaissent. Il est important de noter que les portes d'oubli et de sortie sont utilisées uniquement dans les architectures LSTM. GRU dispose donc de deux portes, alors que LSTM dispose de quatre portes.

2.4.4 Fonctions d'activation

Il existe trois fonctions d'activation qui sont utilisées dans les modèles RNN :

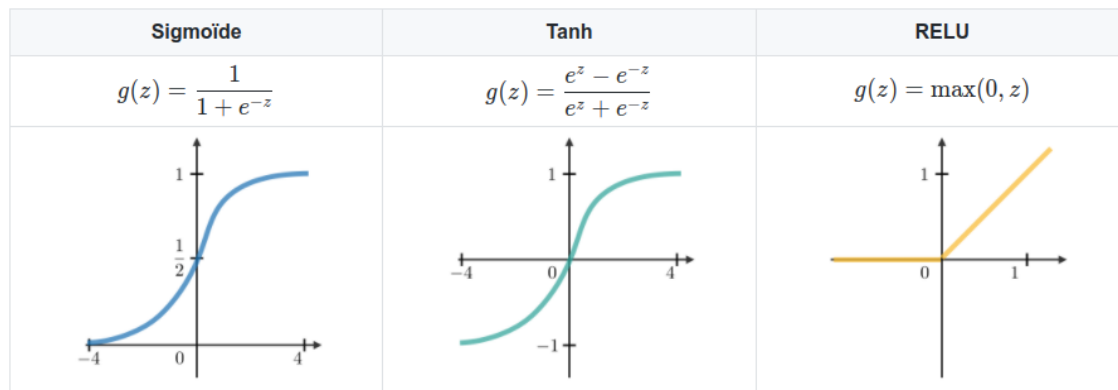


Figure 7: Fonctions d'activation communément utilisées et leurs représentations (Amidi and Amidi 2020)

- La *fonction d'activation sigmoïde* représente la fonction de répartition de la loi logistique, souvent utilisée dans les réseaux de neurones, car elle est dérivable.
- La *fonction d'activation tanh*, ou *tangente hyperbolique*, représente la fonction de répartition de la loi hyperbolique.
- La *fonction d'activation RELU*, ou *rectified linear unit*, représente la fonction de répartition de la loi linéaire.

Le rôle d'une fonction d'activation est de modifier de manière non-linéaire les valeurs de sortie des neurones, ce qui permet de modifier spatialement leur représentation. Une fonction d'activation est donc définie et spécifique pour chaque couche du réseau de neurones. Il ne faut pas confondre avec les fonctions de *loss* qui sont utilisées pour déterminer la qualité de l'apprentissage et sont quant à elles uniques, c'est-à-dire que l'on doit définir une unique fonction de *loss* pour chaque modèle.

2.4.5 GRU et LSTM

Nous pouvons distinguer les unités de porte récurrente (en anglais *Gated Recurrent Unit*) (GRU) et les unités de mémoire à long/court terme (en anglais *Long Short-Term Memory*) (LSTM). Ces deux architectures sont très similaires et visent à atténuer le problème de gradient qui disparaît, rencontré avec les RNNs traditionnels lors de l'apprentissage. *LSTM* peut être vu comme étant une généralisation de *GRU* en utilisant des cellules de mémoire à long ou court terme.

Pour comprendre les différences fondamentales entre les deux architectures, il est nécessaire de regarder en détails les différentes équations utilisées par chacune d'elles.

2.4.5.1 Gated Recurrent Unit (GRU)

Comme nous l'avons vu précédemment, l'architecture GRU comporte deux portes : une porte d'actualisation Γ_u (en anglais *update gate*) et une porte de pertinence Γ_r (en anglais *reset gate*).

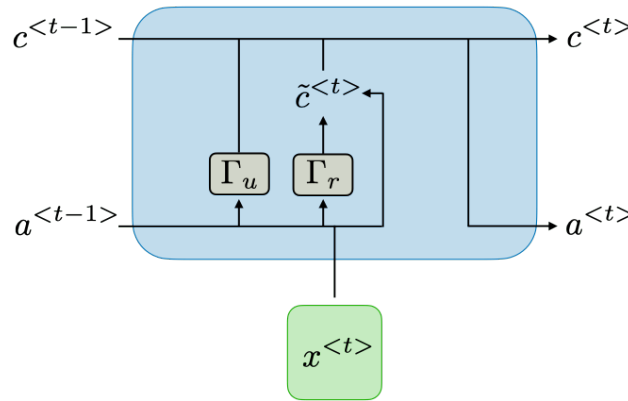


Figure 8: Architecture d'une unité de GRU (Amidi and Amidi 2020)

Il faut discerner trois composantes importantes pour la structure de l'unité de GRU :

- La cellule candidate $c^{<t>}$, où $c^{<t>} = \tanh(W_c[\Gamma_r \star a^{<t-1>}, x^{<t>}] + b_c)$
- L'état final de la cellule $c^{<t>}$, où $c^{<t>} = \Gamma_u \star c^{<t>} + (1 - \Gamma_u) \star c^{<t-1>}$

L'état final de la cellule est calculé par la somme des produits de la porte d'actualisation Γ_u et de la valeur de la cellule candidate $c^{<t>}$ et de l'inverse de la porte d'actualisation $1 - \Gamma_u$ multiplié par la valeur de l'état final de la cellule antérieure $c^{<t-1>}$.

Cet état final de la cellule est donc dépendant de la porte d'actualisation Γ_u et peut soit être mis à jour avec la valeur de la cellule candidate $c^{<t>}$ ou soit conservé la valeur de l'état final de la cellule antérieure.

- La fonction d'activation $a^{<t>}$, où $a^{<t>} = c^{<t>}$

2.4.5.2 Long Short-Term Memory (LSTM)

Maintenant que nous avons vu plus en détails l'architecture générale des RNNs, ainsi que les particularités de GRU, il est temps d'aborder en détails les particularités de l'architecture de LSTM, qui sera l'architecture choisie par le projet final.

En plus des deux portes d'actualisation et de pertinence, LSTM intègre une porte d'oubli Γ_f et une porte de sortie Γ_o .

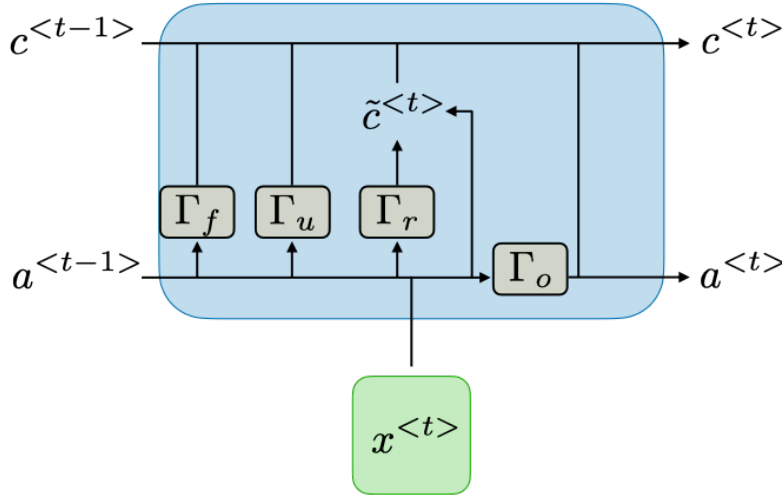


Figure 9: Architecture d'une unité de LSTM (Amidi and Amidi 2020)

L'architecture LSTM est similaire à l'architecture GRU, mais permet de gérer le problème de gradient qui disparaît. Ainsi, aux trois composantes de l'unité de GRU que nous avons vu précédemment, il y a deux différences :

- La fonction d'activation $a^{<t>}$ est désormais multipliée par la porte de sortie Γ_o , ce qui permet de contrôler la quantité d'information qui est libérée par la cellule. On a donc : $a^{<t>} = \Gamma_o \star c^{<t>}$
- L'état final de la cellule $c^{<t>}$ est désormais influencé par la porte d'oubli Γ_f qui devient un facteur de la valeur de l'état final de la cellule antérieure c^{t-1} . En agissant ainsi, la porte d'oubli permet de réguler la quantité d'information retenue de la cellule antérieure. Il y a donc un choix sur ce qui est conservé et oublié. On a ainsi : $c^{<t>} = \Gamma_f \star c^{t-1} + \Gamma_u \star c^{<t-1>}$

2.5 L'avènement des modèles Transformers

Le premier papier de recherche qui présente les modèles Transformers est *Attention Is All You Need* (Vaswani et al. 2017), présenté en juin 2017. L'objectif initial de ce genre d'architecture, qui vient ajouter un mécanisme d'attention aux RNNs, était d'améliorer les performances des modèles existants sur les tâches de traduction en *NLP*.

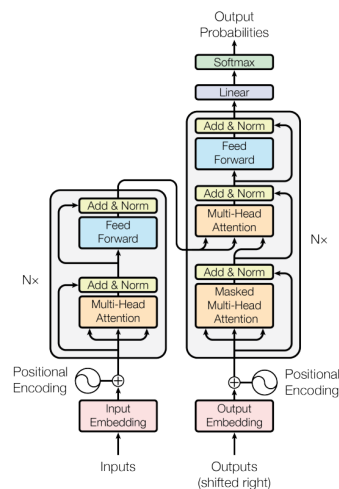


Figure 10: Architecture d'un modèle Transformer (Vaswani et al. 2017)

Nous ne détaillerons pas ici les différentes composantes de cette architecture, mais ce qu'il faut retenir c'est que par l'utilisation d'un encodeur et d'un décodeur, couplés au mécanisme d'attention, il est possible d'entraîner des modèles de manière non-supervisée et surtout d'obtenir des performances encore jamais atteintes par les modèles existants.

L'entraînement de ces modèles de manière non-supervisée est très simple et permet dans un second temps de *finetuner* les modèles avec beaucoup moins de données sur des tâches précises. Ainsi, un même modèle de base peut être entraîné sur différentes tâches et produire des modèles finaux à la pointe des performances des modèles existants.

L'essor de cette architecture a principalement eu lieu dans le domaine du traitement du langage naturel avec des modèles très célèbres tels que *BERT* (Devlin et al. 2018), *GPT-2* (Radford et al. 2019) ou encore *T5* (Raffel et al. 2019). Mais aujourd'hui, il existe également des applications dans les domaines de la vision par ordinateur (*Computer Vision*), des séries temporelles (*Time Series*) ou encore dans le traitement du signal audio (*Audio Processing*).

Cette architecture passionnante est en train de devenir un modèle de référence pour énormément de projet de recherche et cela dans quasiment tous les domaines du Machine Learning. Nous ne détaillerons pas plus ici puisque nous n'avons pas fait usage de cette architecture dans **Make Us Rich**, et cela nécessiterait un dossier à part entière tellement il y a choses à préciser.

Maintenant que nous avons un meilleur aperçu des modèles de référence, et des détails de l'architecture des modèles RNN, et plus particulièrement des modèles LSTM, nous allons pouvoir présenter le projet **Make Us Rich** qui vise à automatiser l'entraînement et le service de modèles LSTM pour de la prédiction sur des données financières.

3 Make Us Rich

Il existe beaucoup de ressources, notamment sur les modèles de prédiction appliqués à des séries temporelles, qui sont très utiles sur des données stationnaires telles que les données météorologiques ou les vols d'avions. En revanche, en ce qui concerne les marchés financiers et les crypto-monnaies, il est difficile de trouver des méthodes dites *classiques* qui soient aussi efficaces et fiables.

Étant passionné par le *Deep Learning* et surveillant d'un oeil positif l'évolution des crypto-monnaies, j'ai souhaité créer un outil permettant de lier les deux. En plus de créer cet outil de prédiction, j'ai voulu aller plus loin et développer le projet pour en faire une base solide à déployer pour quiconque souhaiterait comprendre les enjeux du déploiement de modèles de *Deep Learning* et les outils d'automatisation et de supervision des modèles.

C'est donc avant tout un projet de passionné, mais aussi un formidable outil (en tous cas je l'espère) pour apprendre à automatiser, déployer et mettre à disposition des modèles de *Deep Learning*. Sans plus attendre, je vous invite à découvrir plus en détails le projet et son architecture présentée en introduction de ce dossier.

Pour conserver une certaine lisibilité, les différents blocs de code sont numérotés entre parenthèses et leur détails sont disponibles dans les annexes de ce rapport.

3.1 Les données

La récupération de données et la création d'un jeu de données est la première étape d'un projet de *Machine Learning*, voir même de *Data Science* en général. C'est une étape qui n'est pas à sous-estimer, car elle va déterminer la réussite de votre projet.

3.1.1 Récupération des données

Cette étape de récupération des séries temporelles correspond à l'étape de *fetching* sur le schéma présenté en introduction. Les données que j'utilise sont des données publiques collectées par beaucoup de plateformes de marchés financiers. Pour ce projet, j'ai choisi de récupérer les données de la plateforme *Binance*. *Binance* est une plateforme très connue dans le monde de l'échange de crypto-monnaies et elle dispose d'une API de trading qui permet de récupérer des données très simplement et rapidement. Il suffit de disposer d'un compte sur *Binance* et de se générer un token d'accès pour pouvoir utiliser cette API de façon permanente et sans frais.

Binance dispose également d'un package Python (`python-binance`) qui facilite les appels à son API. J'ai donc codé une classe, `BinanceClient(1)`, qui permet de gérer les interactions avec l'API de *Binance* et qui inclut des méthodes telles que la récupération de données sur cinq jours, un an ou une période à définir par l'utilisateur. Ces méthodes requièrent en argument un symbole de crypto-monnaie et une monnaie de comparaison dans tous les cas et renvoient les données sous forme de `pandas.DataFrame`.

3.1.2 Description des données

Les données récupérées sont des séries temporelles de la crypto-monnaie cible comparée à une monnaie. Par exemple, je peux récupérer les données de la crypto-monnaie *BTC* par rapport à la monnaie *EUR* sur les 5 derniers jours avec un intervalle de 1 heure.

On compte 12 colonnes de données :

- `timestamp` : date et heure de la série temporelle.
- `open` : valeur d'ouverture de la crypto-monnaie cible sur l'intervalle.
- `high` : valeur haute de la crypto-monnaie cible sur l'intervalle.
- `low` : valeur basse de la crypto-monnaie cible sur l'intervalle.
- `close` : valeur de clôture de la crypto-monnaie cible sur l'intervalle.
- `volume` : volume d'échange de la crypto-monnaie cible sur l'intervalle.
- `close_time` : date et heure de la clôture de la crypto-monnaie cible sur l'intervalle.
- `quote_av` (quote asset volume) : correspond au volume d'échange de la monnaie cible sur l'intervalle.
- `trades` : correspond au nombre de trades effectués sur l'intervalle.
- `tb_base_av` (taker buy base asset volume) : correspond au volume d'acheteur de la crypto-monnaie cible sur l'intervalle.
- `tb_quote_av` (taker buy quote asset volume) : correspond au volume d'acheteur de la monnaie cible sur l'intervalle.
- `ignore` : correspond à une colonne qui ne sera pas utilisée.

Ce sont des informations classiques que l'on retrouve souvent sur les plateformes de marchés financiers. Nous allons voir que pour notre cas d'usage, toutes ces données ne seront pas utilisées. De plus, ces données ne seront pas stockées puisqu'elles ne sont plus valables après la fin de l'intervalle de récupération et que le projet prévoit un ré-entraînement toutes les heures des modèles. Ainsi nous n'avons pas besoin de stocker les données pour une utilisation ultérieure, un simple appel à l'API suffit pour récupérer les nouvelles données utiles à un entraînement de modèle.

3.1.3 Préparation des données

Il est nécessaire de préparer les données pour qu'elles soient utilisables par le modèle. Pour cela, nous allons utiliser plusieurs fonctions définies dans l'étape de `preprocessing` de ce projet. C'est à cette étape qu'intervient un choix des *features* à utiliser pour l'entraînement du modèle et leur *engineering*. Le terme *feature engineering* est un terme désignant les différentes étapes de *raffinage* des données pour qu'elles soient utilisables par le modèle.

3.1.3.1 Extraction des données utiles

Tout d'abord, nous allons extraire les données utiles à partir de la série temporelle grâce à la fonction `extract_features_from_dataset()`(2). Pour cela, nous allons utiliser uniquement les colonnes `open`, `high`, `low`, `close` et `timestamp`. Nous stockons également la différence entre la valeur de clôture et la valeur d'ouverture pour chaque intervalle sous le nom `close_change`.

Ainsi à l'issu de cette étape, nous obtenons un nouveau `pandas.DataFrame` qui contient les *features* spécialement sélectionnées pour l'entraînement. Nous n'incluons pas les colonnes relatives aux volumes d'échange et aux trades, car c'est la prédiction de la valeur de clôture sur l'intervalle suivant qui nous intéresse ici. Il serait néanmoins possible d'inclure les notions de volumes dans l'entraînement, mais cela complexifierait le modèle et l'alourdirait pour un gain potentiel à déterminer.

3.1.3.2 Séparation des jeux de données

Une fois nos *features* sélectionnées, nous allons séparer les données en trois jeux de données distincts grâce à la fonction `split_data()`(3). La séparation consiste à diviser les données en deux jeux de données :

- `training_set` : 90%
- `test_set` : 10%

Pour des données temporelles il est important de ne pas mélanger la chronologie des données puisque cela peut créer des problèmes de cohérence. En effet, nous voulons que le modèle puisse prédire les valeurs de clôture sur l'intervalle suivant et non pas sur des intervalles passés.

3.1.3.3 Mise à l'échelle des données

Il est important de mettre à l'échelle les données pour que le modèle puisse les utiliser correctement. Pour cela, nous allons utiliser la fonction `scale_data()`(4). Cette fonction va permettre

de normaliser les données pour qu'elles soient comprises entre -1 et 1 pour nos deux jeux de données. C'est une technique de normalisation qui permet de réduire les écarts entre les données et ainsi les rendre plus facile à manipuler par le modèle lors de l'entraînement.

On utilise ici la méthode de normalisation `MinMaxScaler` de la librairie `sklearn`. Il est important de noter que nous sauvegardons également cet objet de normalisation dans un fichier `pickle` pour pouvoir l'utiliser plus tard lors de l'inférence via l'API. En effet, puisque le modèle est entraîné sur des données normalisées, il est primordial qu'elles le soient également lors des prédictions postérieures. De plus, nous avons besoin de cet objet pour pouvoir inverser la normalisation des données prédites et obtenir des valeurs de clôture réelles, c'est-à-dire des valeurs de clôture non normalisées.

3.1.3.4 Préparation des séquences de données

Il ne reste plus qu'à préparer les données pour qu'elles soient utilisables par le modèle. Pour cela, nous allons devoir créer des séquences de données. Pour cela, nous allons utiliser la fonction `create_sequences()`(5). Cette fonction va utiliser les données préalablement normalisées pour créer des séquences de données de taille `sequence_length`.

C'est à cette étape que nous construisons les *features* d'entrée du modèle et la *target* de sortie, aussi appelé *label*. Dans notre cas, nous utiliserons la colonne `close` comme *target* et le reste des colonnes comme *features*.

Il est à noter que nous allons séparer les données du `training_set` en deux séquences de données distinctes pour avoir également des séquences de données pour la validation du modèle, grâce à la fonction `split_train_and_val_sequences()`(6). Nous utiliserons comme taille `val_size=0.2` pour la validation du modèle, ce qui représente 18% des données totales attribuées pour la validation du modèle.

3.2 Modélisation

Pour ce projet, nous avons choisi d'utiliser un modèle de type *LSTM* pour prédire les valeurs de clôture de la monnaie sur un intervalle de temps de 1 heure. Pour le chargement des données, la définition de l'architecture du modèle ainsi que son entraînement, nous avons choisi d'utiliser la librairie *PyTorch-Lightning* qui est une sur-couche de l'excellente librairie *PyTorch*. Cette librairie permet de packager plus simplement et rapidement du code *PyTorch*, ce qui va nous aider pour le déploiement et le service de nos modèles via notre API dans un second temps.

3.2.1 Définition de l'architecture du modèle

Nous allons commencer par décrire l'architecture du modèle qui se compose de deux parties complémentaires :

- Un premier module `LSTMRegressor(7)` qui définit la structure du modèle, les hyperparamètres, ainsi que les différentes étapes d'inférence via un `nn.Module` de *PyTorch*.
- Un second module `PricePredictor(8)` qui hérite de l'architecture du premier module et qui va permettre de définir les étapes d'entraînement, de validation, de test, le *learning rate* et la fonction de *loss* du modèle.

La fonction de *loss* du modèle est la fonction de coût qui va permettre de déterminer la qualité du modèle. Nous utilisons la fonction de coût `nn.MSELoss()` de la librairie *PyTorch* qui va nous permettre de calculer l'erreur au carré (*mean squared error*, en anglais) entre la valeur prédite et la valeur réelle.

Pour l'entraînement du modèle, nous utiliserons un *dataloader* qui va permettre de charger les données en batchs. C'est la classe `LSTMDataLoader(9)` qui hérite de `CryptoDataset(10)` qui va s'occuper de charger et de distribuer les batchs de données lors des différentes phases d'entraînement, validation et test.

3.2.2 Choix des hyperparamètres

Les hyperparamètres utilisés pour l'entraînement de notre modèle ne sont pas définis dans le code, mais dans un fichier de configuration à part. Cela permet de faciliter la modification des hyperparamètres du modèle et de faciliter le re-entraînement du modèle si besoin. Ils sont donc définis dans un fichier `/conf/base/parameters.yaml` où se trouvent également les paramètres de *fetching* et *preprocessing* des données.

Ce fonctionnement est important puisqu'il permet d'harmoniser le déroulement du pipeline complet. Ainsi, nous n'avons plus besoin de toucher aux fichiers de code pour tester des nouveaux hyperparamètres, de même si nous voulons augmenter la taille des séquences de données.

Voici la liste des hyperparamètres retenus et utilisés pour l'entraînement des modèles :

```
training:
  train_batch_size: 64
  val_batch_size: 1
  train_workers: 2
  val_workers: 1
  max_epochs: 100
  hidden_size: 128
  number_of_features: 9
  number_of_layers: 2
```



```
dropout_rate: 0.2
learning_rate: 1e-4
log_n_steps: 2
run_on_gpu: True # False if running on CPU
wandb_project: "make-us-rich"
```

3.2.3 Entraînements et monitoring

L'entraînement du modèle se fait via la méthode `training_loop()`(11) qui instancie les classes : `LSTMDataloader`, `PricePredictor` utilisées par `Trainer` qui est la classe `Trainer` de *PyTorch-Lightning* qui gère l'entraînement.

Nous utilisons une *seed* pour figer l'aléatoire du modèle, via la fonction `seed_everything` de la librairie *PyTorch-Lightning*, afin de pouvoir reproduire les résultats du modèle si besoin. Nous définissons également deux *callbacks* :

- `ModelCheckpoint()` qui va permettre de sauvegarder les poids du modèle à chaque *epoch* et de conserver uniquement les poids les plus performants.
- `EarlyStopping()` qui va permettre de stopper l'entraînement du modèle si le modèle n'a pas progressé depuis un certain nombre d'*epochs*. Ici, nous utilisons un *patience* de 2 *epochs*.

Dans les deux cas, nous utilisons les valeurs de *loss* de validation, que l'on cherche à minimiser, pour déterminer les poids les plus performants et s'il faut continuer ou stopper l'entraînement.

En ce qui concerne le *monitoring* et le *logging*, nous utilisons la classe `WandbLogger` de *Wandb* incluse dans la librairie *PyTorch-Lightning* qui va nous permettre de stocker les hyperparamètres, l'environnement et toutes les métriques de notre modèle directement sur *Wandb*.

Wandb est un outil de monitoring qui permet de stocker l'historique des entraînements de nos modèles et de comparer les différents modèles. C'est cette plateforme que nous avons privilégié pour l'expérimentation et le monitoring de nos modèles. J'ai donc créé un projet sur *Wandb* pour *Make-Us-Rich* et connecté le pipeline d'entraînement pour que tout soit stocké directement sur *Wandb*(12).

3.2.4 Validation du modèle

La validation du modèle se fait en vérifiant que la moyenne de la valeur de *loss* de validation et celle de test est bien inférieure à une certaine valeur. Cette partie est assez légère et mériterait un ajustement dans notre pipeline d'entraînement automatique. L'architecture du modèle

permet néanmoins d'obtenir des résultats satisfaisants qui sont directement observables et comparables sur l'interface de *Wandb*.

On peut constater tout de même que le taux d'erreur sur les données de validation est vraiment très faible, et il est meilleur que le taux d'erreur sur les données de test. En effet, plus on s'éloigne dans le temps des données d'entraînement, et plus la précision du modèle diminue et donc plus le taux d'erreur, la *loss*, augmente.

3.2.5 Conversion vers ONNX

Nous avons fait le choix d'inclure une étape de conversion automatique du modèle en *ONNX* afin de faciliter la prise en charge de ce modèle par d'autres applications et également un optimisation du temps d'inférence lors du service des modèles via *API*.

En effet le format *ONNX* (*Open Neural Network Exchange*) est un format de représentation de modèle standardisé qui permet, notamment sur *CPU*, de réduire les temps de calcul des modèles (Chaigneau 2022). C'est via deux fonctions que nous allons pouvoir convertir le modèle en *ONNX* et valider que le modèle converti est conforme au modèle original, surtout au niveau de la précision des prédictions.

La première fonction `convert_model()`(13) permet la conversion du modèle en *ONNX* et son stockage avant validation. La seconde fonction `validate_model()`(14) assure que le modèle converti est valable d'un point de vue architecture et noeuds des graphiques, ainsi qu'au niveau de la précision par rapport au modèle *PyTorch* original. La différence entre les deux prédictions doit respecter une tolérance absolue de 10^{-5} et une tolérance relative de 10^{-3} .

3.2.6 Stockage des modèles et des *features engineering*

Il ne nous reste plus qu'à stocker les modèles et les *features engineering* dans une base de données. Vu les données que nous souhaitons conserver, c'est une base de données orientée vers le stockage objet que nous utiliserons, tel que *AWS S3*, *Google Cloud Storage* ou *Azure Blob Storage*. Dans notre cas, nous utilisons *Minio* pour stocker nos données, car c'est l'équivalent de *AWS S3* mais hébergeable n'importe où sur le web ou en local.

C'est grâce à la fonction `upload_files()`(15) que nous allons pouvoir stocker nos modèles et les *features engineering* qui sont associées dans un répertoire unique de notre base de données. Ainsi, ils seront accessibles par la suite par l'API pour leur utilisation. Dans notre cas c'est uniquement le *MinMaxScaler* qui est stocké dans notre base de données au format *pickle*.

Enfin, afin de s'assurer que les fichiers générés par l'entraînement d'un modèle et permettre de conserver de l'espace disque sur la machine qui réalise l'entraînement, nous utilisons une fonction de nettoyage de tous les fichiers locaux qui ne sont plus utilisés. Ceci est réalisé par la dernière fonction du pipeline nommée `clean_files()`[(16)].

3.3 Service des modèles

Maintenant que nous avons terminé l'entraînement du modèle et qu'il est prêt à être utilisé, nous allons créer un service pour le stocker et le rendre accessible via une *API*. C'est exactement la définition du service d'un modèle de *Machine Learning* (*model serving*, en anglais) et c'est une étape cruciale pour permettre à des utilisateurs de bénéficier du produit de *Machine Learning* qu'offre notre service.

3.3.1 Qu'est-ce-que servir des modèles

Servir des modèles de *Machine Learning* est une tâche qui implique la gestion de ressources, de données et le monitoring permanent des performances du modèle. C'est donc une étape à ne pas sous-estimer et qui implique une bonne préparation et une bonne organisation pour réussir.

À cela s'ajoute un facteur d'échelle, qui est la capacité de répondre aux besoins des utilisateurs. Ce n'est pas pareil d'avoir un modèle disponible pour 5 personnes ou pour 50.000 personnes. C'est pourquoi notre approche quant au service de nos modèles de prédictions a été de prévoir la mise à l'échelle en faisant en sorte que le déploiement d'un seul modèle soit identique et répétable pour N modèles et N utilisateurs.

3.3.2 Dockerisation

Pour que le déploiement soit répétable et identique pour chaque modèle et chaque utilisateur, nous utilisons *Docker* comme outil de déploiement. C'est un outil de gestion de conteneurs qui permet de déployer des applications en local ou sur un serveur cloud. On définit une série d'instructions pour la création et le déploiement du ou des différents containers via des fichiers *Dockerfile* et *Docker Compose*.

Le *Dockerfile* est un fichier de configuration qui permet de définir les instructions de création d'un container spécifique, et le *Docker Compose* est un fichier de configuration qui permet de définir les instructions de déploiement de plusieurs containers. Nous avons donc plusieurs fichiers pour l'interface utilisateur(17) et un *Dockerfile* pour l'*API*(18).

3.3.3 Présentation de l'API

Nous allons maintenant décrire l'API, ses différents *endpoints* et leurs rôles. L'avantage de l'API est de pouvoir s'adapter à toutes les exigences de nos utilisateurs. Ainsi, un utilisateur mobile peut demander une prédiction à l'API, tout comme un utilisateur de bureau peut demander une prédiction à l'API via notre interface web ou un script Python. De cette manière, nous pouvons rendre accessible le modèle de prédiction à tous les utilisateurs.

L'API est composée de plusieurs *endpoints*. Chaque endpoint est défini par une URL et une méthode HTTP. Lorsque l'on souhaite accéder à l'API, nous arrivons directement sur la documentation des différents *endpoints*[(19)].

3.3.3.1 Gestion des modèles

Les différents modèles de prédictions sont chargés par l'API grâce à la classe qui les gère, `ModelLoader`(20). Cette classe de gestion du chargement des modèles et de leur prédiction va permettre une flexibilité totale au niveau du nombre de modèle disponible, leur *features engineering* spécifique et leurs informations respectives.

La classe `ModelLoader` se base sur une autre classe `ONNXModel`(21) qui va être le squelette de base pour chacun des modèles de prédictions. Ainsi, cette base permet à chaque modèle de fonctionner de la même manière peu importe la crypto-monnaie sur laquelle il est basé.

Nous avons donc une classe qui permet le fonctionnement de chaque modèle de façon identique et une autre classe qui s'occupe d'orchestrer l'ensemble des modèles de prédictions pour qu'ils soient mis à jour et disponibles pour tous les utilisateurs via les différents endpoints de l'API.

3.3.3.2 Les endpoints de l'API

Les endpoints de l'API sont au nombre de six, avec trois endpoints avec une méthode *PUT* et trois endpoints plus utilitaires disposant d'une méthode *GET*.

- Endpoints de *serving* (22)):
 - `/predict` : permet de récupérer la prédiction d'un modèle d'une crypto-monnaie comparée à une monnaie.
 - `/update_models` : permet de mettre à jour les modèles de prédiction avec les derniers fichiers disponibles dans la base de données.
 - `/update_date` : permet de mettre à jour la date de la dernière mise à jour des modèles, important pour assurer que l'API met toujours à disposition les derniers modèles de prédiction.

- Endpoints de *monitoring* (23):
 - `/check_models_number` : permet de vérifier le nombre de modèles disponibles sur l'*API*.
 - `/healthz` : permet de vérifier le bon fonctionnement de l'*API*. Indispensable si orchestration via *Kubernetes*.
 - `/readyz` : permet de vérifier la disponibilité de l'*API*. Indispensable si orchestration via *Kubernetes*.

3.4 Interface utilisateur

TODO : - présentation de l'interface utilisateur - base de données relationnelle pour authentification - tokens pour appels API

3.5 Packaging du projet

TODO : - ETL - Prefect - Déploiement - Docker, Cloud ou Local - Documentation - Mkdocs material - Alerting

4 Conclusion

TODO : - retour sur projet : ce qui a été fait, reste à faire, mieux, moins bien, ... - ouverture : monétisation, modélisation, fonctionnalités utilisateurs, ...

Annexe 1

```

import pandas as pd

from binance.client import Client
from os import getenv
from typing import Optional

from make_us_rich.utils import load_env

class BinanceClient:

    def __init__(self):
        """
        Initializes the client for connecting to the Binance API.
        """
        try:
            self._config = load_env("binance")
        except:
            self._config = {"API_KEY": getenv("API_KEY"), "SECRET_KEY": getenv("SECRET_KEY")}
            self.client = Client(self._config["API_KEY"], self._config["SECRET_KEY"])
            self.columns = ["timestamp", "open", "high", "low", "close", "volume", "close_time",
                           "quote_av", "trades", "tb_base_av", "tb_quote_av", "ignore"]

    def get_five_days_data(self, symbol: str) -> pd.DataFrame:
        """
        Gets the data for the last five days.

        Parameters
        -----
        symbol: str
            Symbol to get the data for.

        Returns
        -----
        pd.DataFrame
            Dataframe for the last five days.
        """
        symbol = symbol.upper()
        klines = self.client.get_historical_klines(symbol, "1h", "5 day ago UTC")
        data = pd.DataFrame(klines, columns=self.columns)
        data["timestamp"] = pd.to_datetime(data["timestamp"], unit="ms")
        return data

    def get_one_year_data(self, symbol: str) -> pd.DataFrame:
        """
        Gets the data for the last year.

        Parameters
        -----
        symbol: str
            Symbol to get the data for.

        Returns
        -----
        pd.DataFrame
            Dataframe for the last year.
        """
        klines = self.client.get_historical_klines(symbol, "1h", "1 year ago UTC")
        data = pd.DataFrame(klines, columns=self.columns)
        data["timestamp"] = pd.to_datetime(data["timestamp"], unit="ms")
        return data

    def get_data(
        self, symbol: str, interval: str, start_time: str, end_time: Optional[str] = None
    ) -> pd.DataFrame:
        """
        Gets the data for the given symbol, interval, and time range.

        Parameters
        -----
        symbol: str
            Symbol to get the data for.
        interval: str
            Interval to get the data for.
        start_time: str
            Start time of the data.
        end_time: Optional[str]
            End time of the data.

        Returns
        -----
        pd.DataFrame
            Dataframe for the given symbol, interval, and time range.
        """
        klines = self.client.get_historical_klines(symbol, interval, start_time, end_time)
        data = pd.DataFrame(klines, columns=self.columns)
        data["timestamp"] = pd.to_datetime(data["timestamp"], unit="ms")
        return data

```

Annexe 2

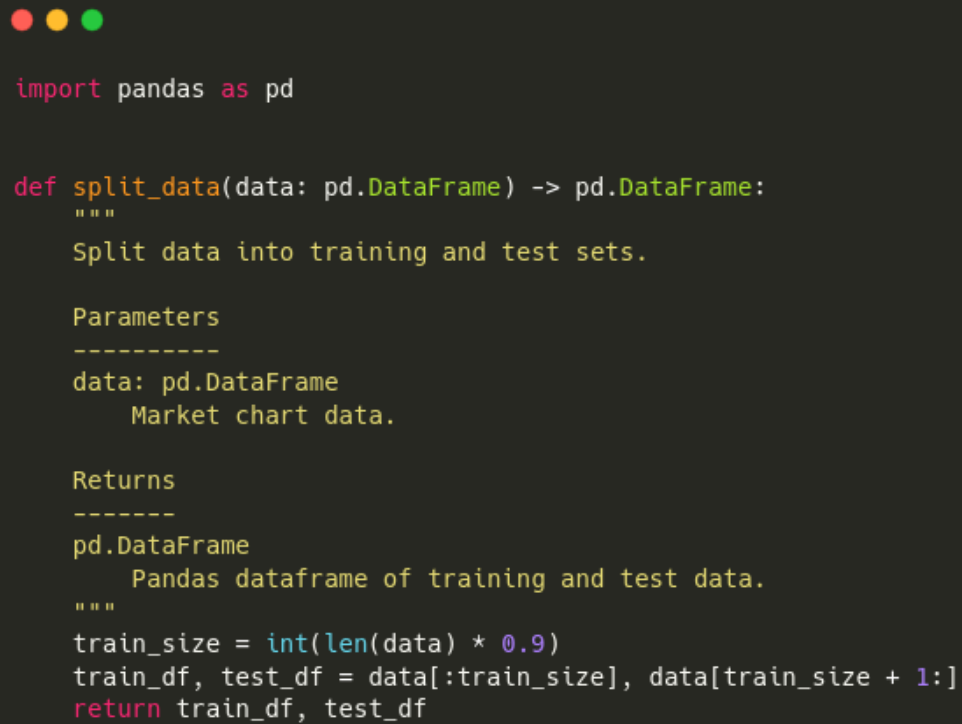
```
import pandas as pd

def extract_features_from_dataset(data: pd.DataFrame) -> pd.DataFrame:
    """
    Extract features from dataset.

    Parameters
    -----
    data: pd.DataFrame
        Market chart data.

    Returns
    -----
    pd.DataFrame
        Pandas dataframe of features.
    """
    rows = []
    for _, row in data.iterrows():
        row_data = dict(
            day_of_week=row["timestamp"].dayofweek,
            day_of_month=row["timestamp"].day,
            week_of_year=row["timestamp"].week,
            month_of_year=row["timestamp"].month,
            open=row["open"],
            high=row["high"],
            low=row["low"],
            close=row["close"],
            close_change=row["close"] - row["open"],
        )
        rows.append(row_data)
    return pd.DataFrame(rows)
```


Annexe 3



```
import pandas as pd

def split_data(data: pd.DataFrame) -> pd.DataFrame:
    """
    Split data into training and test sets.

    Parameters
    -----
    data: pd.DataFrame
        Market chart data.

    Returns
    -----
    pd.DataFrame
        Pandas dataframe of training and test data.
    """
    train_size = int(len(data) * 0.9)
    train_df, test_df = data[:train_size], data[train_size + 1:]
    return train_df, test_df
```

Annexe 4

```
import pandas as pd

from pickle import dump
from sklearn.preprocessing import MinMaxScaler

def scale_data(
    train_df: pd.DataFrame, test_df: pd.DataFrame, dir_path: str,
) -> pd.DataFrame:
    """
    Scale data to have a mean of 0 and a standard deviation of 1.

    Parameters
    -----
    train_df: pd.DataFrame
        Training data.
    test_df: pd.DataFrame
        Test data.
    dir_path: str
        Directory path to save the scaler.

    Returns
    -----
    pd.DataFrame
        Scaled training and test data.
    """
    scaler = MinMaxScaler(feature_range=(-1, 1))
    scaler = scaler.fit(train_df)

    scaled_train_df = pd.DataFrame(
        scaler.transform(train_df),
        index=train_df.index,
        columns=train_df.columns,
    )
    scaled_test_df = pd.DataFrame(
        scaler.transform(test_df),
        index=test_df.index,
        columns=test_df.columns,
    )
    dump(scaler, open(f"{dir_path}/scaler.pkl", "wb"))
    return scaled_train_df, scaled_test_df
```

Annexe 5

```
import pandas as pd

from typing import List, Tuple

def create_sequences(
    input_data: pd.DataFrame,
    target_column: str,
    sequence_length: int,
) -> List[Tuple[pd.DataFrame, float]]:
    """
    Create sequences from the input data.

    Parameters
    -----
    input_data: pd.DataFrame
        Pandas dataframe of input data.
    target_column: str
        Name of the column to predict.
    sequence_length: int
        Length of the sequence.

    Returns
    -----
    List[Tuple[pd.DataFrame, float]]
        List of sequences.
    """
    sequences = []
    size = len(input_data)
    for i in range(size - sequence_length):
        sequence = input_data[i: i + sequence_length]
        label_position = i + sequence_length
        label = input_data.iloc[label_position][target_column]
        sequences.append([sequence, label])
    return sequences
```

Annexe 6

```
def split_train_and_val_sequences(
    sequences: List[Tuple[pd.DataFrame, float]], val_size: float,
) -> Tuple[List[Tuple[pd.DataFrame, float]]]:
    """
    Split sequences into training and validation sets.

    Parameters
    -----
    sequences: List[Tuple[pd.DataFrame, float]]
        List of sequences.
    val_size: float
        Percentage of the data to use as validation.

    Returns
    -----
    Tuple[List[Tuple[pd.DataFrame, float]]]
        Tuple of training and validation sequences.
    """
    train_sequences, val_sequences = [], []
    for sequence, label in sequences:
        if len(train_sequences) < len(sequences) * (1 - val_size):
            train_sequences.append((sequence, label))
        else:
            val_sequences.append((sequence, label))
    return train_sequences, val_sequences
```

Annexe 7

```
import torch.nn as nn

class LSTMRegressor(nn.Module):
    """
    Standard LSTM model with PyTorch Lightning.
    """
    def __init__(self,
                 batch_size: int,
                 dropout_rate: float,
                 hidden_size: int,
                 number_of_features: int,
                 number_of_layers: int,
                 run_on_gpu: bool,
                 ):
        super().__init__()
        self.batch_size = batch_size
        self.dropout_rate = dropout_rate
        self.hidden_size = hidden_size
        self.n_features = number_of_features
        self.number_of_layers = number_of_layers
        self.run_on_gpu = run_on_gpu

        self.lstm = nn.LSTM(
            batch_first=True,
            dropout=self.dropout_rate,
            hidden_size=self.hidden_size,
            input_size=self.n_features,
            num_layers=self.number_of_layers,
        )

        self.regressor = nn.Linear(self.hidden_size, 1)

    def forward(self, x):
        """
        Forward pass through the model.
        lstm_out = (batch_size, sequence_length, hidden_size)
        """
        if self.run_on_gpu:
            self.lstm.flatten_parameters()
        _, (hidden, _) = self.lstm(x)
        out = hidden[-1]
        return self.regressor(out)
```

Annexe 8

```
class PricePredictor(pl.LightningModule):
    """
    Training model with PyTorch Lightning.
    """
    def __init__(self,
                  batch_size: int,
                  dropout_rate: float,
                  hidden_size: int,
                  learning_rate: float,
                  number_of_features: int,
                  number_of_layers: int,
                  run_on_gpu: bool,
                  criterion: nn.Module = nn.MSELoss(),
    ) -> None:
        """
        Initialize the model.

        Parameters
        -----
        batch_size: int
            Batch size for training.
        dropout_rate: float
            Dropout rate for the LSTM.
        hidden_size: int
            Hidden size for the LSTM.
        learning_rate: float
            Learning rate for the optimizer.
        number_of_features: int
            Number of features in the input.
        number_of_layers: int
            Number of layers in the LSTM.
        run_on_gpu: bool
            Whether to run the model on the GPU.
        criterion: nn.Module
            Loss function to use.
        Returns
        -----
        None
        """
        super().__init__()
        self.model = LSTMRegressor(
            batch_size, dropout_rate, hidden_size, number_of_features, number_of_layers, run_on_gpu,
        )
        self.learning_rate = learning_rate
        self.criterion = criterion
        self.save_hyperparameters()

    def forward(self, x, labels=None) -> Tuple[float, torch.Tensor]:
        """
        Forward pass through the model.

        Parameters
        -----
        x: torch.Tensor
            Input data.
        labels: torch.Tensor
            Labels for the data.
        Returns
        -----
        loss: float
            Loss for the model.
        output: torch.Tensor
            Output of the model.
        """
        output = self.model(x)
        if labels is not None:
            loss = self.criterion(output, labels.unsqueeze(dim=1))
            return loss, output
        return output
```

```
def training_step(self, batch: Tuple[torch.Tensor, torch.Tensor], batch_idx: int) -> Dict:
    """
    Training step.

    Parameters
    -----
    batch: Tuple[torch.Tensor, torch.Tensor]
        Tuple of input data and labels.
    batch_idx: int
        Batch index.

    Returns
    -----
    Dict
        Dictionary with the train loss.
    """
    sequences, labels = batch
    loss, _ = self(sequences, labels)
    self.log("train/loss", loss, on_step=True, on_epoch=True)
    return {"loss": loss}

def validation_step(self, batch: Tuple[torch.Tensor, torch.Tensor], batch_idx: int) -> Dict:
    """
    Validation step.

    Parameters
    -----
    batch: Tuple[torch.Tensor, torch.Tensor]
        Tuple of input data and labels.
    batch_idx: int
        Batch index.

    Returns
    -----
    Dict
        Dictionary with the valid loss.
    """
    sequences, labels = batch
    loss, _ = self(sequences, labels)
    self.log("valid/loss", loss, on_step=True, on_epoch=True)
    return {"loss": loss}

def test_step(self, batch: Tuple[torch.Tensor, torch.Tensor], batch_idx: int) -> Dict:
    """
    Test step.

    Parameters
    -----
    batch: Tuple[torch.Tensor, torch.Tensor]
        Tuple of input data and labels.
    batch_idx: int
        Batch index.

    Returns
    -----
    Dict
        Dictionary with the test loss.
    """
    sequences, labels = batch
    loss, _ = self(sequences, labels)
    self.log("test/loss", loss, on_step=True, on_epoch=True)
    return {"loss": loss}

def configure_optimizers(self) -> torch.optim.AdamW:
    """
    Configure the optimizer.

    Returns
    -----
    torch.optim.adamw.AdamW
        Optimizer.
    """
    return torch.optim.AdamW(self.parameters(), lr=self.learning_rate)
```

Annexe 9

```
import pandas as pd
import pytorch_lightning as pl

from torch.utils.data import DataLoader
from typing import List, Tuple

from make_us_rich.pipelines.training import CryptoDataset

class LSTMDataLoader(pl.LightningDataModule):
    """
    Data loader for the LSTM model.
    """
    def __init__(self,
                 train_sequences: List[Tuple[pd.DataFrame, float]],
                 val_sequences: List[Tuple[pd.DataFrame, float]],
                 test_sequences: List[Tuple[pd.DataFrame, float]],
                 train_batch_size: int,
                 val_batch_size: int,
                 train_workers: int = 2,
                 val_workers: int = 1,
    ):
        """
        Initialize the data loader.
        Parameters
        """
        train_sequences: List[Tuple[pd.DataFrame, float]]
        List of training sequences.
        val_sequences: List[Tuple[pd.DataFrame, float]]
        List of validation sequences.
        test_sequences: List[Tuple[pd.DataFrame, float]]
        List of test sequences.
        train_batch_size: int
        Batch size for training.
        val_batch_size: int
        Batch size for validation.
        train_workers: int
        Number of workers for training.
        val_workers: int
        Number of workers for validation.
        """
        super().__init__()
        self.train_sequences = train_sequences
        self.val_sequences = val_sequences
        self.test_sequences = test_sequences
        self.train_batch_size = train_batch_size
        self.val_batch_size = val_batch_size
        self.train_workers = train_workers
        self.val_workers = val_workers
        self.test_workers = val_workers

    def setup(self, stage: str = None) -> None:
        """
        Load the data.
        Parameters
        """
        stage: str
        Name of the stage.
        """
        self.train_dataset = CryptoDataset(self.train_sequences)
        self.val_dataset = CryptoDataset(self.val_sequences)
        self.test_dataset = CryptoDataset(self.test_sequences)

    def train_dataloader(self):
        """Return the training data loader."""
        return DataLoader(
            self.train_dataset,
            batch_size=self.train_batch_size,
            shuffle=False,
            num_workers=self.train_workers
        )

    def val_dataloader(self):
        """Return the validation data loader."""
        return DataLoader(
            self.val_dataset,
            batch_size=self.val_batch_size,
            shuffle=False,
            num_workers=self.val_workers
        )

    def test_dataloader(self):
        """Return the test data loader."""
        return DataLoader(
            self.test_dataset,
            batch_size=self.val_batch_size,
            shuffle=False,
            num_workers=self.test_workers
        )
```


Annexe 10

```
import pandas as pd
import torch

from torch.utils.data import Dataset
from typing import List, Tuple

class CryptoDataset(Dataset):
    """
    Dataset class for the LSTM model used by PyTorch Lightning.
    """
    def __init__(self, sequences: List[Tuple[pd.DataFrame, float]]):
        self.sequences = sequences

    def __len__(self):
        return len(self.sequences)

    def __getitem__(self, index: int):
        sequence, label = self.sequences[index]
        return (torch.Tensor(sequence.to_numpy()), torch.tensor(label).float())
```

Annexe 11

```
import pandas as pd

from pytorch_lightning import Trainer, callbacks, seed_everything
from pytorch_lightning.loggers import WandbLogger
from typing import Any, Dict, List, Tuple

from .model import PricePredictor
from .dataloader import LSTMDataLoader

def training_loop(
    train_sequences: List[Tuple[pd.DataFrame, float]],
    val_sequences: List[Tuple[pd.DataFrame, float]],
    test_sequences: List[Tuple[pd.DataFrame, float]],
    parameters: Dict[str, Any],
    dir_path: str,
):
    """
    Training loop for the LSTM model.

    Parameters
    -----
    train_sequences: List[Tuple[pd.DataFrame, float]]
        List of training sequences.
    val_sequences: List[Tuple[pd.DataFrame, float]]
        List of validation sequences.
    test_sequences: List[Tuple[pd.DataFrame, float]]
        List of test sequences.
    parameters: Dict[str, Any]
        Hyperparameters for the model.
    dir_path: str
        Path to the directory where the model will be saved.
    """
    seed_everything(42, workers=True)
    logger = WandbLogger(project=parameters["wandb_project"])
    gpu_value = 1 if parameters["run_on_gpu"] is True else 0

    model = PricePredictor(
        batch_size=parameters["train_batch_size"],
        dropout_rate=parameters["dropout_rate"],
        hidden_size=parameters["hidden_size"],
        learning_rate=parameters["learning_rate"],
        number_of_features=parameters["number_of_features"],
        number_of_layers=parameters["number_of_layers"],
        run_on_gpu=parameters["run_on_gpu"],
    )

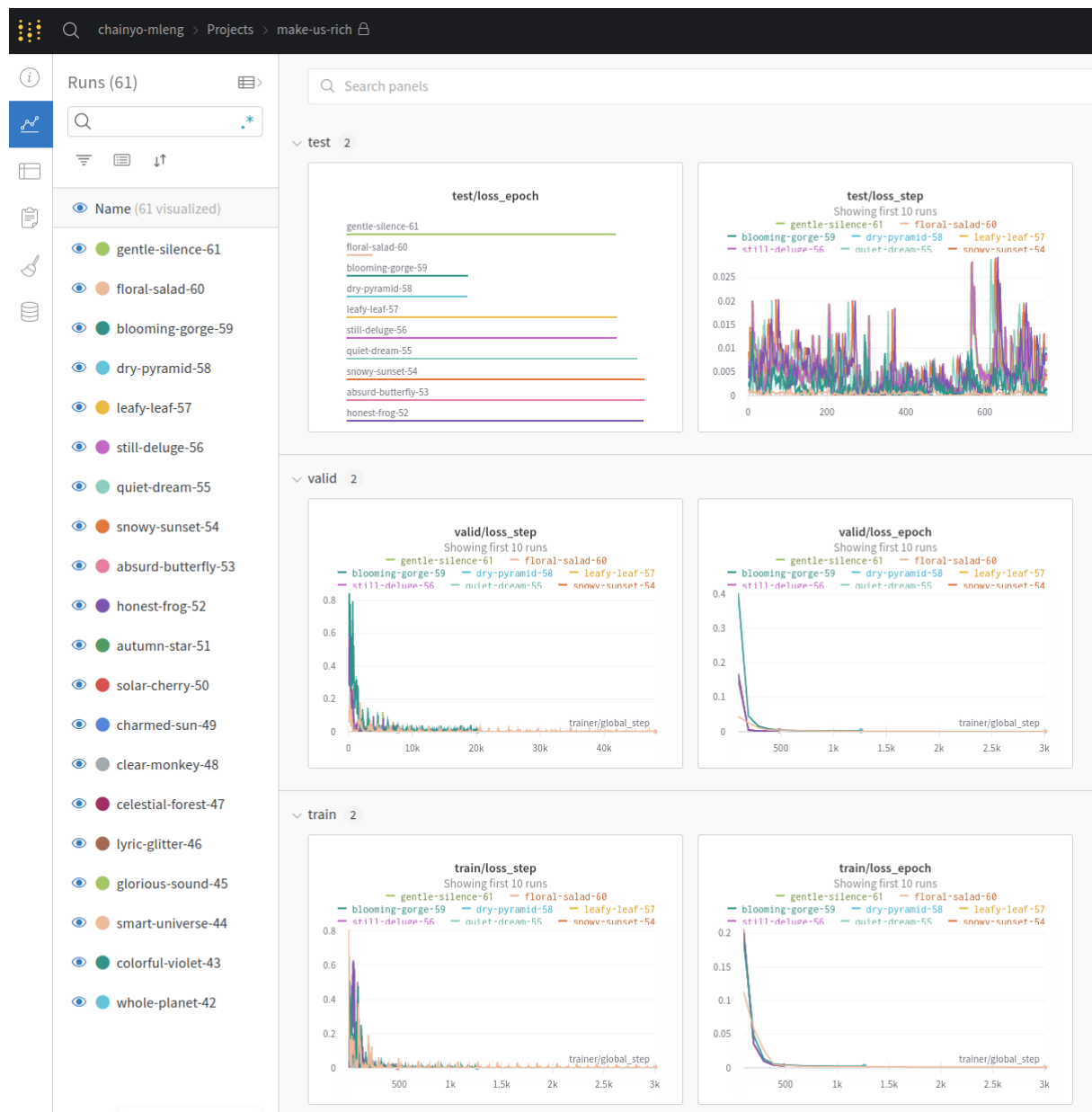
    data_module = LSTMDataLoader(
        train_sequences=train_sequences,
        val_sequences=val_sequences,
        test_sequences=test_sequences,
        train_batch_size=parameters["train_batch_size"],
        val_batch_size=parameters["val_batch_size"],
        train_workers=parameters["train_workers"],
        val_workers=parameters["val_workers"],
    )

    checkpoint_callback = callbacks.ModelCheckpoint(
        dirpath=dir_path,
        save_top_k=1,
        verbose=True,
        monitor="valid/loss",
        mode="min",
    )
    early_stopping_callback = callbacks.EarlyStopping(
        monitor="valid/loss",
        patience=2,
        verbose=True,
        mode="min",
    )

    trainer = Trainer(
        max_epochs=parameters["max_epochs"],
        logger=logger,
        callbacks=[checkpoint_callback, early_stopping_callback],
        gpus=gpu_value,
        log_every_n_steps=parameters["log_n_steps"],
        progress_bar_refresh_rate=10,
        deterministic=True,
    )
    trainer.fit(model, data_module)
    trainer.test(model, data_module)

    return {"training_done": True}
```

Annexe 12



Annexe 13

```

def convert_model(
    train_sequences: List[Tuple[pd.DataFrame, float]],
    val_sequences: List[Tuple[pd.DataFrame, float]],
    test_sequences: List[Tuple[pd.DataFrame, float]],
    parameters: str,
    dir_path: str,
    training_done: Dict[str, bool],
) -> Dict[str, Any]:
    """
    Convert trained model to ONNX.

    Parameters
    -----
    train_sequences: List[Tuple[pd.DataFrame, float]]
        Training sequences.
    val_sequences: List[Tuple[pd.DataFrame, float]]
        Validation sequences.
    test_sequences: List[Tuple[pd.DataFrame, float]]
        Test sequences.
    parameters: str
        Parameters used for training.
    dir_path: str
        Directory path where the model is saved.
    training_done: Dict[str, bool]
        Flag indicating if the training is done.
    Returns
    -----
    Dict[str, Any]
        Dictionary of outputs from the conversion step.
    """
    if training_done["training_done"] == True:
        model_path = [file for file in glob.glob(f"{dir_path}/*.ckpt")][0]
        model = PricePredictor.load_from_checkpoint(model_path)
        data = LSTMDataLoader(
            train_sequences=train_sequences,
            val_sequences=val_sequences,
            test_sequences=test_sequences,
            train_batch_size=parameters["batch_size"],
            val_batch_size=parameters["batch_size"],
        )
        data.setup()
        input_batch = next(iter(data.train_dataloader()))
        input_sample = input_batch[0][0].unsqueeze(0)
        path_onnx_model = f"{dir_path}/model.onnx"
        torch.onnx.export(
            model, input_sample, path_onnx_model,
            export_params=True,
            opset_version=11,
            input_names=["sequence"],
            output_names=["output"],
            dynamic_axes={
                "sequence": {0: "batch_size"},
                "output": {0: "batch_size"},
            },
        )
    return {
        "conversion_done": True,
        "model_path": model_path,
        "input_sample": input_sample,
    }

```

Annexe 14

```
def validate_model(
    dir_path: str,
    conversion_outputs: Dict[str, Any],
) -> Dict[str, bool]:
    """
    Check if the converted model is valid.

    Parameters
    -----
    dir_path: str
        Directory path where the model is saved.
    conversion_outputs: Dict[str, Any]
        Dictionary of outputs from the conversion step.

    Returns
    -----
    Dict[str, bool]
        Flag indicating if the model is valid.
    """
    if conversion_outputs["conversion_done"] == True:
        path_onnx_model = f"{dir_path}/model.onnx"
        onnx_model = onnx.load(path_onnx_model)
        try:
            onnx.checker.check_model(onnx_model)
        except onnx.checker.ValidationError as e:
            raise ValueError(f"ONNX model is not valid: {e}")

        try:
            input_sample = conversion_outputs["input_sample"]
            model = PricePredictor.load_from_checkpoint(conversion_outputs["model_path"])
            model.eval()
            with torch.no_grad():
                torch_output = model(input_sample)
            ort_session = onnxruntime.InferenceSession(path_onnx_model)
            ort_inputs = {ort_session.get_inputs()[0].name: to_numpy(input_sample)}
            ort_outputs = ort_session.run(None, ort_inputs)
            np.testing.assert_allclose(to_numpy(torch_output), ort_outputs[0], rtol=1e-03, atol=1e-05)
            print("🎉 ONNX model is valid. 🎉")
        except Exception as e:
            raise ValueError(f"ONNX model is not valid: {e}")
    return {"validation_done": True}
```

Annexe 15

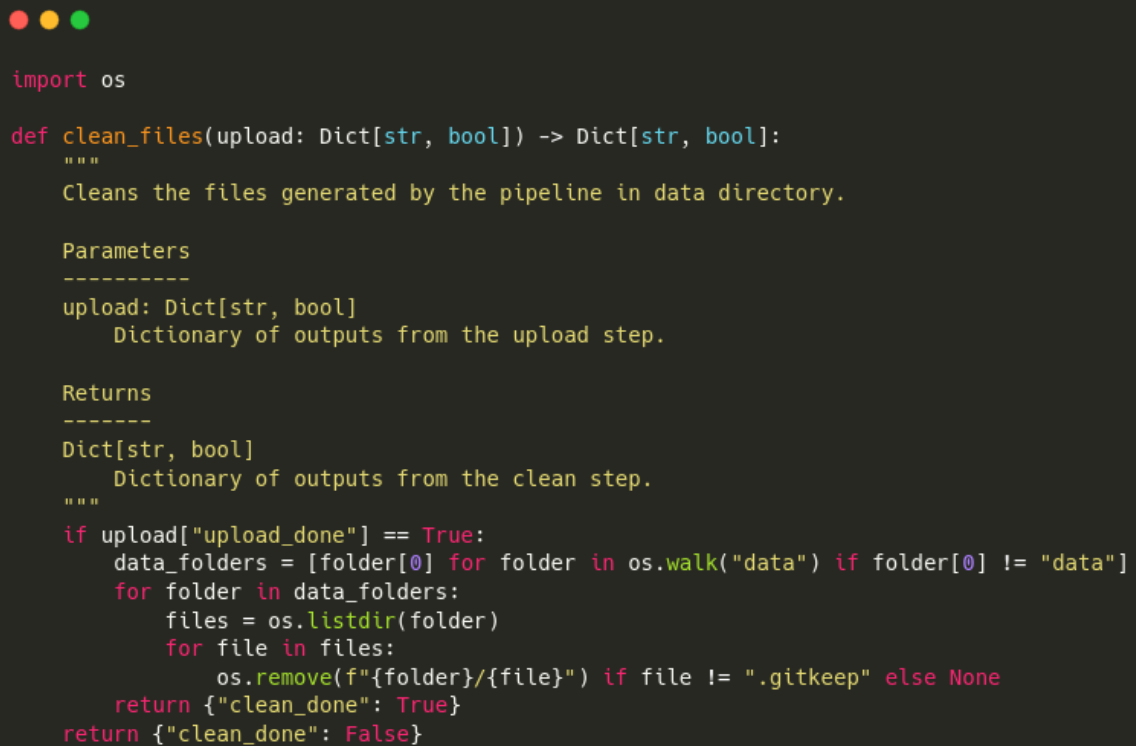
```
from datetime import datetime
from typing import Dict, Tuple

from make_us_rich.client import MinioClient

def upload_files(
    currency: str,
    compare: str,
    dir_path: str,
    validation: Dict[Tuple[str, bool], Tuple[str, str]],
) -> Dict[str, bool]:
    """
    Uploads model and features engineering files to Minio.

    Parameters
    -----
    currency: str
        Currency used in the model.
    compare: str
        Compare used in the model.
    validation: Dict[Tuple[str, bool], Tuple[str, str]]
        Dictionary of outputs from the validation step.
    dir_path: str
        Directory path where the model files are saved.
    Returns
    -----
    Dict[Tuple[str, bool]]
        Dictionary of outputs from the upload step.
    """
    if validation["validation_done"] == True:
        client = MinioClient()
        date = datetime.now().strftime("%Y-%m-%d")
        model_path = f"{dir_path}/model.onnx"
        client.upload(client.bucket, f"{date}/{currency}_{compare}/model.onnx", model_path)
        scaler_path = f"{dir_path}/scaler.pkl"
        client.upload(client.bucket, f"{date}/{currency}_{compare}/scaler.pkl", scaler_path)
        return {"upload_done": True}
    return {"upload_done": False}
```

Annexe 16

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The code is written in Python and defines a function named 'clean_files'. The function takes a dictionary 'upload' as input and returns a dictionary. It includes docstrings for the function's purpose, parameters, and returns. The logic involves checking if 'upload_done' is True, then walking through the 'data' directory, listing files, and removing them except for '.gitkeep'.

```
import os

def clean_files(upload: Dict[str, bool]) -> Dict[str, bool]:
    """
    Cleans the files generated by the pipeline in data directory.

    Parameters
    -----
    upload: Dict[str, bool]
        Dictionary of outputs from the upload step.

    Returns
    -----
    Dict[str, bool]
        Dictionary of outputs from the clean step.
    """
    if upload["upload_done"] == True:
        data_folders = [folder[0] for folder in os.walk("data") if folder[0] != "data"]
        for folder in data_folders:
            files = os.listdir(folder)
            for file in files:
                os.remove(f"{folder}/{file}") if file != ".gitkeep" else None
        return {"clean_done": True}
    return {"clean_done": False}
```

Annexe 17

```
FROM python:3.8

WORKDIR /code
COPY ./requirements.txt /code/requirements.txt

RUN pip install --no-cache-dir -r requirements.txt

COPY ./app.py /code/app/app.py

CMD ["bash", "-c", "streamlit run app/app.py"]
```

```
version: "3.8"

services:
  interface:
    image: mur-interface
    build:
      context: ./
      dockerfile: Dockerfile
    restart: unless-stopped
    container_name: mur-interface
    environment:
      - HOST=${HOST}
      - NAME=${NAME}
      - USER=${USER}
      - PWD=${PWD}
      - URL=${URL}
    ports:
      - "8502:8501"

  postgres:
    container_name: postgres
    image: postgres:13.4
    environment:
      POSTGRES_USER: $USER
      POSTGRES_PASSWORD: $PWD
      POSTGRES_DB: $NAME
    volumes:
      - ./database/init.sql:/docker-entrypoint-initdb.d/init.sql
      - ./database/postgres-data:/var/lib/postgresql/data
    ports:
      - "5432:5432"
    restart: unless-stopped

  pgadmin:
    container_name: pgadmin
    image: dpage/pgadmin4:snapshot
    environment:
      PGADMIN_DEFAULT_EMAIL: $PGADMIN_EMAIL
      PGADMIN_DEFAULT_PASSWORD: $PGADMIN_PWD
      PGDATA: /var/lib/postgresql/data
    volumes:
      - pgadmin-data:/var/lib/pgadmin
    ports:
      - "5050:80"
    restart: unless-stopped
    depends_on:
      - postgres

volumes:
  pgadmin-data:
```


Annexe 18



```
FROM python:3.8

WORKDIR /code
COPY ./requirements.txt /code/requirements.txt

RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt

COPY ./main.py /code/app/main.py

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "80"]
```

Annexe 19

Annexe 19

Annexe 20

```

class ModelLoader:
    """
    Loader class for interacting with the Minio Object Storage API.
    """
    def __init__(self):
        self.client = MinioClient()
        self.session_models = {}
        self.storage_path = Path.cwd().joinpath("api", "models")
        self.update_date()
        self.update_model_files()

    def get_predictions(self, model_name: str, sample: pd.DataFrame) -> float:
        """
        Gets the predictions from the model.

        Parameters
        -----
        model_name: str
            Name of the model.
        sample: pd.DataFrame
            Sample to predict.

        Returns
        -----
        float
            Predicted value.
        """
        if self._check_model_exists_in_session(model_name):
            model = self.session_models[model_name]["model"]
            return model.predict(sample)
        else:
            raise ValueError("Model not found in session.")

    def update_date(self):
        """
        Updates the date of the loader.
        """
        self.date = datetime.now().strftime("%Y-%m-%d")

    def update_model_files(self):
        """
        Updates the model files in the serving models directory.
        """
        for model in self._get_list_of_available_models():
            currency, compare = model.split("-")
            self._download_files(currency, compare)
            self._add_model_to_session_models(currency, compare)

    def _get_models_files_path(self, currency: str, compare: str):
        """
        Returns the path to the files in models directory.

        Parameters
        -----
        currency: str
            Currency used in the model.
        compare: str
            Compare used in the model.

        Returns
        -----
        str
            Path to the model files.
        """
        model = self.storage_path.joinpath(f"{currency}_{compare}", "model.onnx")
        scaler = self.storage_path.joinpath(f"{currency}_{compare}", "scaler.pkl")
        return model, scaler

    def _mkdir(self, currency: str, compare: str) -> None:
        """
        Creates a directory for the model files if it doesn't exist.

        Parameters
        -----
        currency: str
            Currency used in the model.
        compare: str
            Compare used in the model.
        """
        self.storage_path.joinpath(f"{currency}_{compare}").mkdir(exist_ok=True)

```

```
def _download_files(self, currency: str, compare: str) -> None:
    """
    Downloads model and features engineering files from Minio.

    Parameters
    -----
    currency: str
        Currency used in the model.
    compare: str
        Compare used in the model.
    """
    self._mkdir(currency, compare)
    self.client.download(
        self.client.bucket,
        f"{self.date}/{currency}_{compare}/model.onnx",
        f"{self.storage_path}/{currency}_{compare}/model.onnx"
    )
    self.client.download(
        self.client.bucket,
        f"{self.date}/{currency}_{compare}/scaler.pkl",
        f"{self.storage_path}/{currency}_{compare}/scaler.pkl"
    )

def _get_list_of_available_models(self) -> List[str]:
    """
    Looks for available models in the Minio bucket based on the date.

    Returns
    -----
    List[str]
        List of available models.
    """
    available_models = self.client.list_objects(
        self.client.bucket, prefix=self.date, recursive=True
    )
    return list(set([model.object_name.split("/")[1] for model in available_models]))

def _add_model_to_session_models(self, currency: str, compare: str) -> str:
    """
    Adds a new model to the model session.

    Parameters
    -----
    currency: str
        Currency used in the model.
    compare: str
        Compare used in the model.

    Returns
    -----
    str
    """
    model_path, scaler_path = self._get_models_files_path(currency, compare)
    model = OnnxModel(model_path=model_path, scaler_path=scaler_path)
    self.session_models[f"{currency}_{compare}"] = {"model": model}
    return f"Model {model} added to session."

def _check_model_exists_in_session(self, model_name: str) -> bool:
    """
    Checks if the model exists in the current session.

    Parameters
    -----
    model_name: str
        Name of the model.

    Returns
    -----
    bool
    """
    if model_name in self.session_models.keys():
        return True
    return False
```

Annexe 21

```

class OnnxModel:
    def __init__(self, model_path: PosixPath, scaler_path: PosixPath):
        self.model_path = model_path
        self.scaler_path = scaler_path
        self.model_name = self.model_path.parent.parts[-1]
        self.model = onnxruntime.InferenceSession(str(model_path))
        self.scaler = self._load_scaler()
        self.descaler = self._create_descaler()

    def __repr__(self) -> str:
        return f"<OnnxModel: {self.model_name}>"

    def predict(self, sample: pd.DataFrame) -> float:
        """
        Predicts the close price based on the input sample.

        Parameters
        -----
        sample: pd.DataFrame
            Input sample.
        Returns
        -----
        float
            Predicted close price.
        """
        X = self._preprocessing_sample(sample)
        inputs = {self.model.get_inputs()[0].name: to_numpy(X)}
        results = self.model.run(None, inputs)[0][0]
        return self._descaling_sample(results)

    def _create_descaler(self) -> MinMaxScaler:
        """
        Creates a descaler.

        Returns
        -----
        MinMaxScaler
        """
        descaler = MinMaxScaler()
        descaler.min_, descaler.scale_ = self.scaler.min_[:-1], self.scaler.scale_[:-1]
        return descaler

    def _descaling_sample(self, sample) -> None:
        """
        Descalings the sample.

        Parameters
        -----
        sample: numpy.ndarray
            Sample to be descaled.

        Returns
        -----
        float
            Descaled sample.
        """
        values_2d = np.array(sample)[:-1, np.newaxis]
        return self.descaler.inverse_transform(values_2d).flatten()

    def _load_scaler(self) -> MinMaxScaler:
        """
        Loads the scaler from the model files.

        Returns
        -----
        MinMaxScaler
        """
        with open(self.scaler_path, "rb") as file:
            return load(file)

    def _preprocessing_sample(self, sample: pd.DataFrame) -> torch.tensor:
        """
        Preprocesses the input sample.

        Parameters
        -----
        sample: pd.DataFrame
            Input sample.

        Returns
        -----
        torch.tensor
            Preprocessed sample.
        """
        data = extract_features_from_dataset(sample)
        scaled_data = pd.DataFrame(
            self.scaler.transform(data), index=data.index, columns=data.columns
        )
        return torch.Tensor(scaled_data.values).unsqueeze(0)

```

Annexe 22

```

@app.put("/predict", include_in_schema=True, tags=["serving"])
async def predict(currency: str, compare: str, token: str = None):
    """
    Predict endpoint.

    Parameters
    -----
    currency: str
        Currency used in the model.
    compare: str
        Compare used in the model.
    token: str
        API token of the user.
    """
    if token is None:
        return {
            "error": "You need to provide an API token. Check docs for more information."
        }
    model_name = f"{currency}_{compare}"
    symbol = ".".join(model_name.split("_"))
    data = client.get_five_days_data(symbol)
    response = models.get_predictions(model_name, data)
    return {"data": data.to_dict(), "prediction": float(response)}

@app.put("/update_models", include_in_schema=True, tags=["serving"])
async def update_model():
    """
    Update models endpoint.
    """
    models.update_model_files()
    return {"message": "All models have been updated."}

@app.put("/update_date", include_in_schema=True, tags=["serving"])
async def update_date():
    """
    Update date endpoint.
    """
    models.update_date()
    return {"message": "Date has been updated."}

```

Annexe 23

```

@app.get("/check_models_number", include_in_schema=True, tags=["monitoring"])
async def check_models_number():
    """
    Check models number endpoint.
    """
    number_of_running_models = len(models.session_models)
    if number_of_running_models == 0:
        return {"message": "Warning: No models are running."}
    else:
        response = {
            "message": f"Number of running models: {number_of_running_models}",
            "models": [],
        }
        for model in models.session_models:
            response["models"].append(model)
        return response

@app.get("/healthz", status_code=200, include_in_schema=True, tags=["monitoring"])
async def healthz():
    """
    Healthz endpoint.
    """
    return {"status": "ok"}

@app.get("/readyz", status_code=200, include_in_schema=True, tags=["monitoring"])
async def readyz():
    """
    Readyz endpoint.
    """
    return {"status": "ready"}

```

Annexe 24

Annexe 24

Annexe 25

Annexe 25

- Amidi, Afshine, and Shervine Amidi. 2020. "Recurrent Neural Networks Cheatsheet Star." CS 230 - *Recurrent Neural Networks Cheatsheet*. Stanford.edu. <https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks>.
- Chaigneau, Thomas. 2022. "Boost Any Machine Learning Model with ONNX Conversion." *Medium*. Towards Data Science. <https://medium.com/towards-data-science/boost-any-machine-learning-model-with-onnx-conversion-de34e1a38266>.
- Chailan, Romain, and F. Palacios-Rodríguez. 2018. "TP-Timeseries Novembre 2018." *Rchailan.github.io*. https://rchailan.github.io/assets/lectures/timeseries/tp_timeseries.html.
- Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding." arXiv. <https://doi.org/10.48550/ARXIV.1810.04805>.
- Goude, Yannig. 2020. "Les Processus Arima." Paris-Saclay. https://www.imo.universite-paris-saclay.fr/~goude/Materials/time_series/cours6_ARIMA.pdf.
- Kafritsas, Nikos. 2021. *Temporal Fusion Transformer: Time Series Forecasting with Interpretability*. <https://towardsdatascience.com/temporal-fusion-transformer-googles-model-for-interpretable-time-series-forecasting-5aa17beb621>.
- Lim, Bryan, Serkan O. Arik, Nicolas Loeff, and Tomas Pfister. 2019. "Temporal Fusion Transformers for Interpretable Multi-Horizon Time Series Forecasting." arXiv. <https://doi.org/10.48550/ARXIV.1912.09363>.
- Nielsen, Aileen. 2019. *Practical Time Series Analysis*. O'Reilly Media.
- Radford, Alec, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. *Language Models Are Unsupervised Multitask Learners*. OpenAI. <https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf>.
- Raffel, Colin, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2019. "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer." arXiv. <https://doi.org/10.48550/ARXIV.1910.10683>.
- Rumelhart, Geoffrey E, David E; Hinton, and Ronald J Williams. 1985. "Learning Internal Representations by Error Propagation," September. <https://apps.dtic.mil/dtic/tr/fulltext/u2/a164453.pdf>.
- Sherstinsky, Alex. 2020. "Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network." *Physica D: Nonlinear Phenomena* 404 (March): 132306. <https://doi.org/10.1016/j.physd.2019.132306>.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. “Attention Is All You Need.” arXiv. <https://doi.org/10.48550/ARXIV.1706.03762>.