Projet Recherche

Analyse comparative de Python et de Julia pour le développement d'une plateforme d'apprentissage

Nom Prénom Numéro : SERRAF Dan 3971120
M1 Informatique DAC – UE Projet Année : 2021/2021
Encadrant : MARSALA Christophe

Sommaire

1) Introduction

Dans ce projet, je cherche à réaliser une analyse comparative entre 2 langages de programmation dans le domaine du Machine Learning. Je comparerai d'une part le Python qui est largement utilisé par tous depuis plus de 30 ans et permet de coder des applications Web jusqu'aux systèmes d'IA avancé. D'autre part, nous avons le Julia qui est un récent langage de haut niveau, performant et dynamique pour le calcul scientifique.

Je commencerai par réaliser une comparaison sur l'aspect bibliographique pour mettre en évidence les librairies d'apprentissage automatique existantes pour ces 2 langages.

Puis je ferai une recherche sur les aspects liés à la programmation en termes de conception de solutions qui prendra en compte le style de code et l'architecture générale du programme.

Enfin, je vais créer des librairies pour l'apprentissage automatique pour chacun d'eux, je comparerai les performances en temps d'exécution de différents algorithmes de Machine Learning sur des jeux de données d'apprentissage.

2) Comparatif python et Julia

A) Langages

Python est un langage de programmation interprété multiparadigme. Il favorise la programmation impérative structurée, et orientée objet. Il est doté d'un typage dynamique fort, d'une gestion automatique de la mémoire par ramasse-miettes et d'un système de gestion d'exceptions.

Il est conçu pour optimiser la productivité des programmeurs en offrant des outils de haut niveau et une syntaxe simple à utiliser.

Python est un langage qui peut s'utiliser dans de nombreux contextes et s'adapter à tout type d'utilisation grâce à des bibliothèques spécialisées à chaque traitement. Il est cependant particulièrement utilisé comme langage de script pour automatiser des tâches simples mais fastidieuses.

On l'utilise également comme langage de développement de prototype lorsqu'on a besoin d'une application fonctionnelle avant de l'optimiser avec un langage de plus bas niveau. Il est particulièrement répandu dans le monde scientifique, et possède de nombreuses extensions destinées aux applications numériques.

Julia est un langage de programmation de haut niveau, performant et dynamique pour le calcul scientifique, avec une syntaxe familière aux utilisateurs d'autres environnements de développement similaires. Il fournit un compilateur sophistiqué, un système de types dynamiques avec polymorphisme paramétré, une exécution parallèle distribuée, des appels directs de fonctions C, Fortran et Python.

La bibliothèque, essentiellement écrite dans le langage Julia lui-même, intègre également des bibliothèques en C et Fortran pour l'algèbre linéaire, la génération des nombres aléatoires, les transformer de Fourier et le traitement de chaînes de caractères. Les programmes Julia sont organisés autour de la définition de fonctions, et de leur surcharge autour de différentes combinaisons de types d'arguments. Toutefois la particularité de Julia se trouve plus dans l'analyse numérique et à la science des données. Elle prend également en charge l'informatique concurrente, parallèle et distribuée qui est indispensable avec les quantités de données qui augmente de manière exponentielle.

B) Bibliothèques

Dans le domaine de la science des données, on retrouve différent type de bibliothèques. Les bibliothèques qui permettent d'extraire les données, de les traiter et les modéliser, d'utiliser des modèles existants et enfin les bibliothèques pour visualiser les données ainsi que nos résultats.

En ce qui concerne les bibliothèques pour l'extraction des données, on retrouve principalement Scrapy et BeautifulSoup pour python et Gumbo pour Julia. Ces bibliothèques aident à construire des programmes d'exploration qui permettent de récupérer des données structurées sur le web tel que des Url, formulaires ou autre.

Une fois les données extraites, viennent les bibliothèques qui permettent de les traiter et de les modéliser. Pour les calculs scientifiques et la réalisation d'opérations, optimiser sur les tableaux on retrouve Numpy pour python. Pour Julia, ils sont déjà intégrés par défaut et on n'a pas besoin de librairies externes. Ils existent également d'autres structures de données comme des données tabulaires ou bien encore d'autres structures particulières comme les matrices clairsemées. Nous retrouvons l'ensemble de ces structures particulières dans Python et Julia à travers Scipy. Puis pour les données tabulaires ont les retrouves pour Python dans la célèbre bibliothèque Pandas et pour Julia dans la bibliothèque Dataframe.

Suite à cela, on utilise différents modèles de Machine Learning ou Deep Learning afin d'extraire des informations sur nos données en fonction du problème recherché (classification, régression, réduction dimensions …). Les bibliothèques les plus populaires sont Sklearn pour le Machine Learning et Tensorflow, Pytorch et Keras pour le Deep Learning.

Pour terminer on visualise nos données et nos résultats à l'aide de diagramme et de graphique plus avancés. La plus populaire est la bibliothèque Matplotlib pour Python et Plot pour Julia.

On remarque que pour la majorité des bibliothèques sous Python il existe son équivalence en Julia. Cela est possible grâce à une bibliothèque Julia, Pycall qui permet d'exécuter du code python sous Julia. Ainsi une grande majorité des librairies python sont utilisable directement sous Julia.

C) Architecture générale

Le paradigme de programmation est la façon d'approcher la programmation informatique et de formuler les solutions aux problèmes et leur formalisation dans un langage de programmation approprié. En ce qui concerne Python et Julia, peuvent approcher les paradigmes de programmation impérative et fonctionnelle. La différence a lieu au niveau du paradigme de programmation Object et au niveau du typage. Le python prend en charge la programmation Object alors que le Julia ne le prend pas. Toutefois il existe une notion d'héritage à travers les types mais au même niveau qu'un langage de programmation objet car on ne peut pas joindre une méthode à une structure. Les méthodes ne peuvent être déclarées qu'en dehors de la structure. Enfin le Julia prend en compte le typage statique et dynamique alors que python prend en compte uniquement le typage dynamique. Cette particularité différencie Julia des autres langages de programmation.

D) Avantages et inconvénients

A présent on va présenter les différents avantages de chaque langage de programmation.

L'avantage majeur de Julia est la compilation, non interprétée, ce qui le rend rapide**.** Julia utilise le Framework LLVM (Low Level Virtual Machine) pour la compilation JIT (Just-In-Time), ce qui lui permet d'offrir la même vitesse de runtime que le C. Python peut être rendu plus rapide grâce à des bibliothèques externes, des compilateurs JIT tiers (PyPy) et des optimisations avec des outils comme Cython, mais Julia est conçue pour être plus rapide dès le départ.

Un élément intéressant est la syntaxe, est optimisée pour les mathématiques**.** Le langage Julia a une syntaxe optimisée pour les mathématiques et les langages ou environnements scientifiques, ce qui facilite la compréhension des non-programmeurs.

Un autre point qu'on peut soulever, est qu'il existe des bibliothèques d'apprentissage automatique natives développées sur Julia tel que Flux.jl par exemple. Flux est une bibliothèque d'apprentissage automatique pour Julia qui contient de nombreux modèles existants pour des cas d'utilisation courants. Une bibliothèque écrite en Julia afin que la modification soit simple, en utilisant la compilation JIT de Julia pour optimiser les projets.

Pour terminer, Julia a l'avantage de prendre en compte des interfaces de fonctions étrangères. Julia peut s'interfacer avec des bibliothèques externes comme celles de Python, C et Fortran. Il est également possible de s'interfacer avec du code Python via la bibliothèque PyCall, et même de partager des données entre Python et Julia.

D'un autre coté, nous avons le python qui possède une popularité reconnue dans le monde entier. En effet une langue n'est rien sans une communauté nombreuse, dévouée et active autour d'elle. Julia a une communauté enthousiaste qui ne cesse de grandir, mais elle est encore loin de la communauté de Python qui représente des millions d'utilisateurs.

Un autre point intéressant, est que python possède une plus large variété de packages tiers sur différents sujets. Python profite d'une plus large variété de packages tiers pour le Machine Learning, tandis que très peu de logiciels tiers sont développés autour de Julia à l'heure actuelle.

De plus la mise en route moins coûteuse et plus rapide car le runtime de Python est plus léger que celui de Julia ainsi, il faut généralement moins de temps aux programmes Python pour démarrer et fournir les premiers résultats. De plus, alors que la compilation JIT accélère le temps d'exécution des programmes Julia, elle se fait au prix d'un démarrage plus lent.

Pour terminer, le python comme tous les langages de programmation ont l'habitude de commencer leur indexation a 0 par opposition au langage comme Julia qui commence leurs indexations à 1 qui peut être source de problèmes.

3 Expérimentation

Pour que la comparaison entre python et Julia soit le plus juste possible, il faut d'une part que les expériences ont lieu sur les mêmes données. D'autre part il faut que les modèles aient été créés de la même manière pour les différents langages de programmation.

J'ai réalisé mes campagnes d'expériences dans un premier temps sur des bases de données de petites tailles tel que le jeu de donnée des Iris. Ce jeu de données a été utilisé dans le débogage de mes modèles mais également pour valider l'exactitude de ces derniers. Suite à cela, pour évaluer mes modèles je les ai utilisés sur le jeu de donnée Usps qui est un jeu de données plus conséquent. Pour terminer, j'ai utilisé le jeu de donnes Mnist afin de voir de manière plus accentuée la différence entre python et Julia pour des gros jeux de données, pour des algorithmes de Machine Learning qui coûte cher en complexité mémoire, complexité de calcul ou bien encore un algorithme non supervisé.

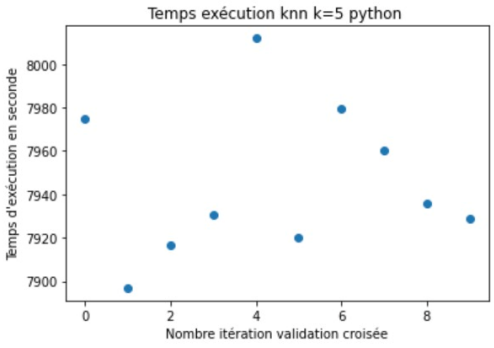
Les expériences ont été réalisées par validation croisée avec 10 itérations. Les données de train test ont été générées depuis python puis exportées sous format Excel, afin de travailler sur les mêmes données sur Julia. Dans le cadre de ce projet de recherche, nous ne recherchions pas à optimiser les modèles pour avoir les meilleurs accuracy possibles. De ce fait nous fixons les hyperparamètres de nos modèles avec des hyperparamètres qui donne de résultat correct. Le but recherché est de montrer que nous obtenons des résultats similaires entre Python et Julia mais également des résultats corrects qui montre que nos algorithmes marchent.

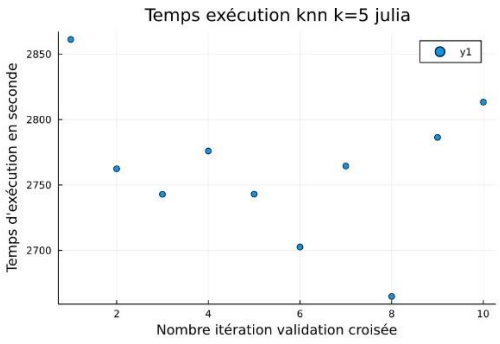
Les graphiques qui suivront pour les différents modèles de machine learning ont été réalisés sur le jeu de données mnist. Elle regroupe 60000 images d'apprentissage et 10000 images de test. Ce sont des images en noir et blanc, normalisées centrées de 28 pixels de côté.

A) K-plus-proche voisin

Notre première comparaison se fera sur l'algorithme des k plus proche voisins. Nous avons choisi de comparer ces 2 langages sur ce premier algorithme car il est couteux en mémoire et en temps d'exécution.

Nous pouvons voir dans les graphiques ci-dessous l'algorithme des k-plus-proches voisins sous python et Julia.





On obtient pour un nombre de voisin égale à 5, une moyenne et un écart type de 0.97 et 0.004 pour l'accuracy, ce qui nous montre que les modèles sont juste.

En ce qui concerne le temps d'exécution, on obtient un temps moyen de 2 heures 12 minutes et 26 secondes ainsi qu'un écart type de 33 secondes pour python. Et on obtient pour Julia, un temps moyen d'exécution de 46 min et 2 secondes ainsi qu'un écart type de 54 secondes.

Cette première expérimentation nous montre que le Julia est presque 3 fois rapide sur cet algorithme de Machine Learning pour les mêmes implémentations du modèle. Un élément intéressant à relever, est que pour la première exécution Julia est légèrement plus lente. Cette différence est due à la compilation. Toutefois même au démarrage, on remarque que le Julia est beaucoup plus efficace que le python.

Nous allons à présent, observer l'architecture de code entre le python et Julia pour voir si on a la même facilité d'écriture, afin de faire un affichage concis du code, j'ai volontairement supprimé la documentation.

```python
class ClassifierSupervised(ABC):
    @abstractmethod
    def __init__(self):
        raise NotImplementedError("Merci d'implémenter cette méthode.")
    @abstractmethod
    def fit(self, X, y):
        raise NotImplementedError("Merci d'implémenter cette méthode.")
    @abstractmethod
    def predict(self, X):
        raise NotImplementedError("Merci d'implémenter cette méthode.")
    @abstractmethod
    def accuracy(self, X, y):
        raise NotImplementedError("Merci d'implémenter cette méthode.")

class ClassifierBaseSupervised(ClassifierSupervised):
    def __init__(self):
        self.X = np.array([])
        self.y = np.array([])

    def fit(self, X, y):
        self.X = X
        self.y = y
        return self

    def accuracy(self, X, y):
        return 0 if X.shape[0] == 0 else np.array([(self.predict(Xi) == y[i])
        / X.shape[0] for i,Xi in enumerate(X)]).sum()

class ClassifierKNN(ClassifierBaseSupervised):
    def __init__(self, k=5,p=2 ):
        super().__init__()
        self.k = k
        self.p = p

    def getDistance(self,X):
        return np.linalg.norm(self.X - X,axis=1)

    def getVoisin(self,X):
        return self.y[np.argsort(self.getDistance(X))[0:self.k]]

    def predict(self, X_test):
        y_pred = self.getVoisin(X_test)
        return np.bincount(y_pred).argmax()
```

```julia
mutable struct ClassifierKNN
    X::Matrix{Float64}
    y::Vector{Int64}
    k::Int64
    p::Int64
end

function initKnn(X::Matrix{Float64} , y::Vector{Int64} , k::Int64 ,p::Int64  )
    return ClassifierKNN(X, y, k, p)
end

function fit!(Knn::ClassifierKNN,  X::Matrix{Float64}, y::Vector{Int64})
    Knn.X = X
    Knn.y = y
    return Knn
end

function getDistance(Knn::ClassifierKNN, X::Vector{Float64})
    dist = sum(x -> x^2, Knn.X .- reshape(X,1,size(X)[1]); dims=2)
    dist .= sqrt.(dist)
    return dist[:]
end

function getVoisin(Knn::ClassifierKNN, X::Vector{Float64})
    return Knn.y[sortperm(getDistance(Knn,X))[1:Knn.k]]
end

function predict(Knn::ClassifierKNN, X::Vector{Float64})
    y_pred = getVoisin(Knn,X)
    return sb.mode(y_pred)
end

function accuracy(Knn::ClassifierKNN, X::Matrix{Float64}, y::Vector{Int64})
    return size(X)[1] == 0 ? 0 : sum([(predict(Knn,X[i,:]) .== y[i])
     / size(X)[1] for i in 1:size(X)[1] ])
end
```

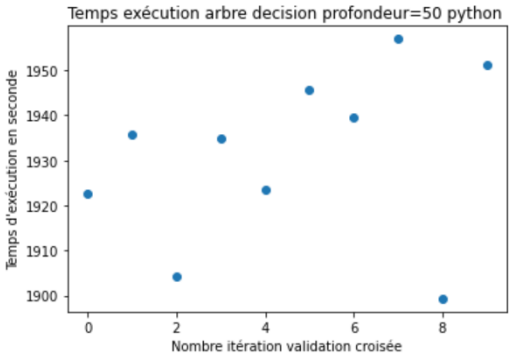
On observe une première différence entre ces derniers en ce qui concerne l'architecture de code. Nous avons pour python une programmation objet (code gauche) et une programmation fonctionnelle (code droite) pour le Julia. Mais lorsque on se penche sur le code, on remarque une forte similarité entre eux et une facilité de codage pour le Julia. Pour coder sous Julia, il faut tout d'abord se familiariser avec le concept de Vector et de Matrix qui sont un peu les équivalents des Array en 1d et 2d sous Numpy puis dans un second temps chercher les fonctions équivalentes de Numpy sous Julia comme par exemple sortperm qui est l'équivalent de argsort. Cette première expérience nous encourage dans notre approfondissement de cette étude pour ce langage.

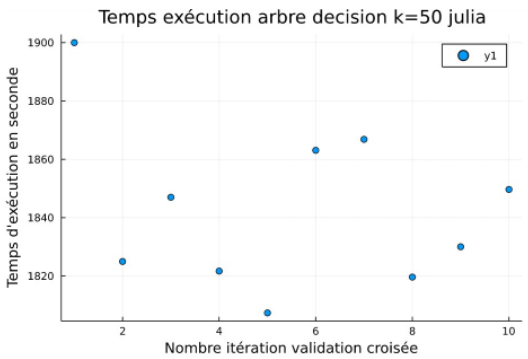
B) Arbre de décision

Notre seconde comparaison se fera sur les arbres de décision. Nous avons choisi ce dernier car c'est un algorithme très utilisé en particulier pour gagner en explication des données. De plus, il possède une structure plus complexe, il permettra de mettre en avant la difficulté de programmation du langage Julia et voir s'il reste beaucoup plus performant que le python.

Pour l'implémentation de l'arbre de décision, j'ai utilisé l'entropie de Shannon comme critère de séparation. De plus pour les données numériques, je sépare les données en deux sous-groupes en fonction d'une valeur seuil. Enfin, il est possible de régler la profondeur de l'arbre à l'aide d'un paramètre max_profondeur.

Nous pouvons voir dans les graphiques ci-dessous l'algorithme d'arbre de décision sous python et Julia.





On obtient pour profondeur maximal égale à 50 une moyenne et un écart type de 0.86 et 0.001 pour l'accuracy, ce qui nous montre que les modèles sont encore une fois corrects.

Cette fois en ce qui concerne le temps d'exécution on obtient un temps moyen de 32 minutes et 11 secondes ainsi qu'un écart type de 18 secondes pour python. Et on obtient pour Julia un temps moyen d'exécution de 30 min et 17 secondes ainsi qu'un écart type de 42 secondes.

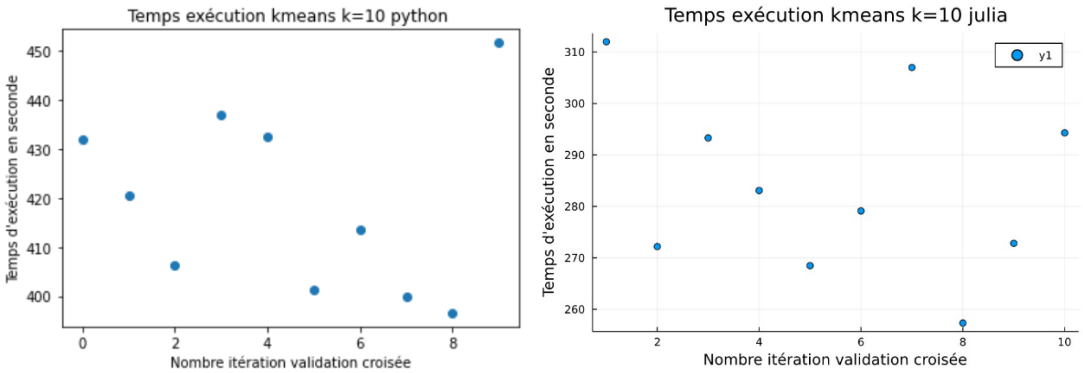
Pour ce modèle, on remarque que malheureusement les exécutions sont équivalentes, cela s'explique par le fait que pour l'arbre de décision j'ai fait le choix de l'implémenter pour qu'il prenne en compte à la fois les attributs catégoriels et à la fois numérique (à l'opposition par exemple de sklearn qui ne prend en compte que les donnes numériques). Ce choix a pour conséquence que les signatures des fonctions ne sont pas autant simplifiées que pour le knn. J'ai donc laissé le compilateur compiler à la volé les différents types en fonction des données, ce qui implique un temps d'exécution plus lent.

Nous pouvons voir dans les graphiques ci-dessous l'algorithme d'arbre de décision sous python et Julia.

C) Kmeans

Notre dernière expérience a été réalisée sur l'algorithme des kmeans. Le choix pour cet algorithme a été fait afin de parcourir l'ensemble des principaux types de modèle d'apprentissage. Nous aurons vu d'une part des algorithmes supervisés tel que les k plus proches voisins et les arbres de décisions et d'autre part le kmeans qui est un algorithme non supervisé.

En ce qui concerne l'évaluation de l'algorithme des kmeans puisque nous n'avons pas d'étiquette, nous n'allons pas calculer l'accuracy mais à la place, la pureté du modèle. La pureté consiste à calculer la classe majoritaire d'un cluster et de dire que les autres éléments du cluster sont mal étiquetés. Dans notre cas, pour simplifier nous n'avons pris que 10 clusters faisant références aux 10 classes de chiffre. Bien qu'en général la méthode est de prendre un nombre de cluster bien supérieur aux nombres de classes puis de fusionner ces derniers, ici encore une fois on veut juste vérifier qu'on a des résultats cohérents.



Pour cette dernière expérience, on obtient pour un nombre de cluster égale à 10, une pureté moyenne de 0.8 et un écart type de 0.02.

Le temps d'exécution pour Julia pour l'algorithme est légèrement plus rapide que celui de python presque 2 fois plus rapide. On obtient une moyenne de temps d'exécution de 4 minutes et 44 secondes pour Julia contre 6 minutes et 59 secondes pour Python.

Encore une fois ici, je pense que la différence se fait au niveau de la programmation Julia, pour ce modèle j'ai pu typer les différentes variables ce qui nous permet d'obtenir un temps d'exécution légèrement meilleur que python. Toutefois je ne suis malheureusement pas encore expert en Julia et je pense qu'il manque des petites améliorations au niveaux technique

4) Pour aller plus loin

Pour aller plus loin dans mes expériences j'ai jugé intéressant de comparer les différents algorithmes de Sklearn sous Python et sous Julia à l'aide de la bibliothèque Pycall afin de voir si le fait d'importer nos modèles sur Julia directement aller améliorer nos performances.

Avant d'analyser les temps d'exécution de ces derniers, nous allons rapidement analysé le code pour exécuter la bibliothèque Sklearn sous Julia.

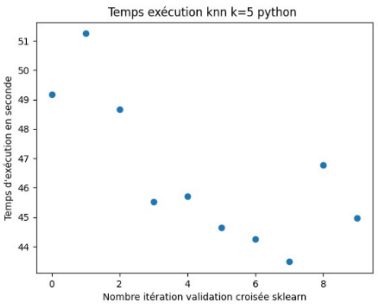
```python
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=k_voisin)
knn = knn.fit(train_x,train_y)
y2 = knn.predict(test_x)
acc = (y2 == test_y).mean()
```

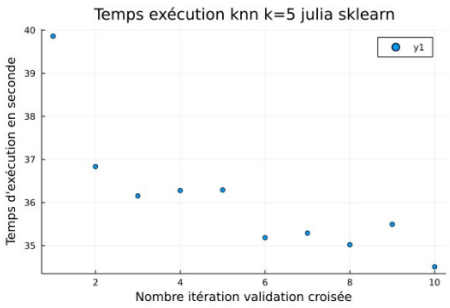
```julia
import PyCall as pc
kn = pc.pyimport("sklearn.neighbors")
knn = kn.KNeighborsClassifier(n_neighbors=k_voisin)
knn.fit(train_x,train_y)
y2 = knn.predict(test_x)
acc = st.mean(y2 .== test_y)
```

Pour importer une bibliothèque Python sous Julia, il suffit d'importer le module PyCall et d'importer les bibliothèques que nous souhaitons utiliser. Ici dans notre cas, j'ai pris l'exemple du k plus proche voisin et on remarque que les codes sont similaires à l'exception de l'importation.

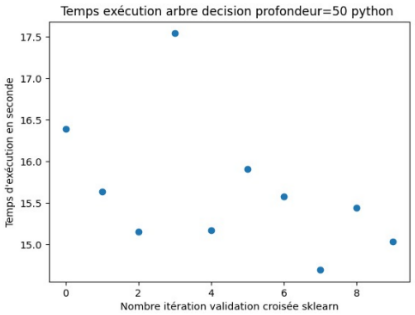
A présent, nous allons passer à l'analyse des temps d'exécution des modèles entre sklearn sous python et sous Julia.

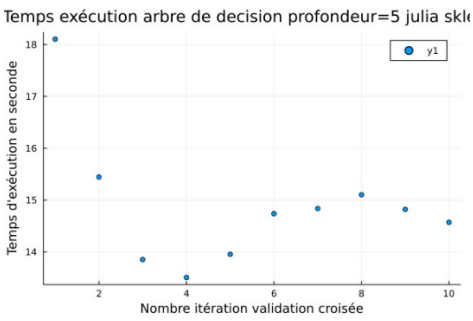
K plus proche voisins



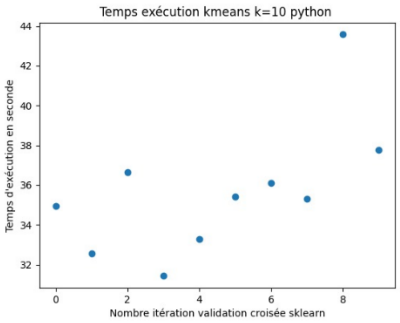


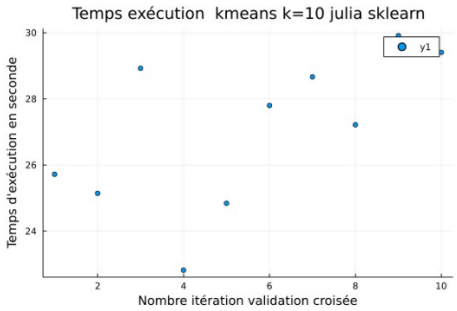
Arbre décision





Kmeans





Tableaux récapitulatif temps exécution moyens sklearn

Temps exprimé en seconde	Python	Julia
K-Plus proche voisin	46	36
Arbre de décision	16	15
Kmeans	36	27

De ces expériences, on peut relever quelques points intéressants. Le premier est que nous obtenons exactement les mêmes résultats en accuracy ce qui implique qu'on peut reproduire notre environnement python directement sous python mais avec un temps d'exécution plus rapide.

Un second élément qui apparait de cette expérience est que lorsque le temps d'exécution est très court, la différence du temps de compilation se fait plus ressentir à travers les temps d'exécution.

Un dernier point pertinent est le temps d'exécution entre les modèles de sklearn et les miens. Cette différence m'a permis de m'interroger sur la qualité de mon code. Après avoir chercher, j'ai trouvé la différence entre mon code et le leur. Mise à part le fait qu'ils réalisent leur code avec des bibliothèques précompilées tel que Cython, il se trouve que leurs modèles sont vectorisés. J'ai donc réalisé une version vectorisée pour le modèle des k plus proches voisins qui m'a permis de me rapprocher des résultats de sklearn.

5) Conclusion

Pour conclure, on retiendra que sur l'aspect bibliographique, il reste encore du travail à faire sur le jeune langage Julia par rapport à python et sa grande communauté et ses nombreuses librairies diverses et variées. Toutefois, ces inconvénients sont très vite comblés par ses nombreux avantages tel que l'importation de fonctions étrangère, çà facilite de codage mais surtout sa compilation qui lui permet de gagner en rapidité.

Du coté de l'implémentation, le Julia propose la programmation fonctionnelle mais pas la programmation objet. Mon expérience personnelle avec le Julia m'a convenu car je suis familier avec la programmation fonctionnelle et objet, les deux me conviennent. Toutefois, il pourrait causer un freinage pour les développeurs, habitués uniquement à la programmation objet.

En ce qui concerne les implémentations des modèles de Machines Learning j'ai pu implémenter l'ensemble des algorithmes demandé sous Python et Julia. Suite à cela, j'ai mené mes campagnes d'expériences qui m'ont permis de confirmer l'hypothèse que l'on a vu au début de ce projet que le Julia est plus rapide que le python. On a toutefois apporté quelques précisions sur les conditions de codage sous Julia afin d'avoir un code rapide, le détail de mes observations sera apporté en annexe.

6) Bibliographie

[1] S. Borağan Aruoba and Jesús Fernández-Villaverde. 2015. A comparison of programming languages in macroeconomics. Journal of Economic Dynamics and Control 58, (September 2015), 265–273. DOI:https://doi.org/10.1016/j.jedc.2015.05.009

[2] Viktor Axillus. 2020. Comparing Julia and Python : An investigation of the performance on image processing with deep neural networks and classification. Retrieved February 27, 2022 from http://urn.kb.se/resolve?urn=urn:nbn:se:bth-19160

[3] Tyler A. Cabutto, Sean P. Heeney, Shaun V. Ault, Guifen Mao, and Jin Wang. 2018. An Overview of the Julia Programming Language. In Proceedings of the 2018 International Conference on Computing and Big Data (ICCBD '18), Association for Computing Machinery, New York, NY, USA, 87–91. DOI: https://doi.org/10.1145/3277104.3277119

[4] Dan Segal (dan@seg.al). Julia Packages. Julia Packages. Retrieved February 27, 2022 from https://juliapackages.com/

[5] White T. The Need for Speed: Julia Vs. Python. Retrieved from https://www.researchgate.net/profile/Mason-White5/publication/357825090_The_Need_for_Speed_Julia_Vs_Python/links/61e109f4c5e3103375918089/The-Need-for-Speed-Julia-Vs-Python.pdf

[6] 2019. Top 10 Python Libraries For Data Science for 2022. Simplilearn.com. Retrieved February 27, 2022 from https://www.simplilearn.com/top-python-libraries-for-data-science-article

[7] 2020. Python vs Julia : quel est le meilleur langage pour la Data Science ? Formation Data Science | DataScientest.com. Retrieved February 27, 2022 from https://datascientest.com/python-vs-julia-quel-est-le-meilleur-langage-pour-la-data-science

[8] 2021. Python vs Julia : comparaison de ces langages pour la Data Science. Infogene. Retrieved February 27, 2022 from https://www.infogene.fr/actualite-blog-expert/python-julia/

[9] 2021. Julia Libraries | Top Julia Machine Learning Libraries. Analytics Vidhya. Retrieved February 27, 2022 from https://www. analyticsvidhya.com/blog/2021/05/top-julia-machine-learninglibraries/

[10] 2021. Les bibliothèque Python à utiliser pour le machine learning. Mobiskill. Retrieved February 27, 2022 from https://mobiskill.fr/blog/conseils-emploi-tech/les-bibliotheque-python-autiliser-pour-le-machine-learning

7) Annexe

A) Tableaux de synthèse

	Python	**Julia**
Scrapping	Scrapy.py BeautifulSoup.py	Gumbo.jl
Structure	Numpy.py Scipy.py Pandas.py	Scipy.jl DataFrame.jl SparseArray.jl
Machine Learning	Sklearn.py	Sklearn.jl
Deep Learning	Tensorflow.py Pytorch.py Keras.py	Tensorflow.jl Pytorch.jl Keras.jl
Visualisation	Matplotlib.py	Plot.jl

Tableaux synthèses bibliothèques

	Python	**Julia**
Impérative	OUI	OUI
Fonctionnelle	OUI	OUI
Object	OUI	NON
Typage	OUI	OUI

Tableaux synthèse paradigme de programmation

	Python	Julia
Open source	OUI	OUI
Langage haut niveau	OUI	OUI
Compiler	NON	OUI
Parallélisme	OUI	OUI
Dynamiquement typée	OUI	OUI
Statiquement typée	NON	OUI
Syntaxe optimisée math	NON	OUI
Interface étrangère	NON	OUI
Grande communauté	OUI	NON
0 – Indexing	OUI	NON
Gérer volume Big data	NON	OUI

Tableaux synthèse avantage et inconvénient

B) Difficultés rencontrées

Lors de ce projet j'ai rencontré plusieurs difficultés majeures.

La première difficulté, a été lors de l'apprentissage du langage Julia, il m'a été difficile de comprendre et nuancer les notations et l'utilisation des Vector et des Matrix ainsi que l'utilisation d'opérateur vectorisé.

La seconde, a été dans le langage lui-même. Il y a des petites habitudes de programmation python qui m'ont fait perdre un peu de temps de débogage au niveau notamment par exemple de l'indexation qui commence à 1 et non pas zéro.

Un élément qui a été particulièrement difficile, a été la mise en place des recherches bibliographiques. En effet, ce projet est récent et malheureusement il n'existe pas encore d'articles scientifiques qui ont réalisé un état de l'art au niveau bibliographique et au niveau de conception de solution.

J'ai également rencontré une difficulté à construire un modèle d'arbre de décision qui marche à la fois pour les données catégorielles et numériques qui prennent en compte également les valeurs continues.

Pour terminer, j'ai eu un peu de mal malheureusement à cause de problèmes techniques dùes à mon ordinateur qui m'a ralenti dans mon travail.

C) Retour expérience

Après avoir passé plus de 3 mois à avoir appris ce langage je pense que le langage Julia a beaucoup de potentiel. Sa facilité de code fait de lui un atout majeur pour des personnes débutantes. Il possède également des atouts qui pourraient intéresser des programmeurs expérimentés comme notamment le fait de typer les objets qui lui permettent de gagner en rapidité de calcule.

Un autre point très intéressant que je trouve au Julia est que nous pouvons également coder avec la syntaxe propre à d'autre langage tel que par exemple le C ou le python directement dans Julia sans pratiquement aucun changement surtout que l'importation des modèles sur Julia apporte un gain de temps. Toutefois il possède encore quelques points à améliorer tel que des méthodes qui sont actuellement manquantes et une communauté encore trop petite. Mais je pense qu'avec le temps c'est 2 points seront corrigés.

D) Optimisation pour Julia

Voici quelques conseils que j'ai pu trouver à travers mes recherches et expériences.

- Pas utiliser de variable globale
- Typer les objets (éviter type abstrait)
- Diviser les fonctions en petite fonctions
- Ecrire des fonctions stables (retourner toujours le même type)
- Eviter de changer le type d'une variable
- Initialiser les tableaux si on connait à l'avance les dimensions
- Accéder aux tableaux par ordre mémoire, le long des colonnes
- Utiliser la syntaxe vectorielle le plus souvent possible