UVSQ université PARIS-SACLAY | UFR des Sciences CAMPUS DE VERSAILLES

RAPPORT DE PROJET TER

Étude d'un algorithme de Machine Learning

Membres du groupe

ELGHRICI Aymene

MANSOURA Mohamed Amine

NAMOUCHI Hamza

Projet TER

Enseignants encadrants : Luc Bouganim et Ludovic Javet

M1 Informatiques 2021/2022

Étape 01 : choix du modèle

- **Choix de l'algorithme**

Après recherche sur la bibliothèque Scikit-Learn et sur Internet, on s'est mis d'accord de choisir l'algorithme de Machine Learning **Stochastic Gradient Descent SGD.**

Lien Scikit-Learn : 1.5. Stochastic Gradient Descent — scikit-learn 1.0.2 documentation
Lien Wikipédia : https://en.wikipedia.org/wiki/Stochastic_gradient_descent
Lien Internet: Stochastic Gradient Descent — Clearly Explained !! | by Aishwarya V Srinivasan | Towards Data Science
Ce choix est motivé par plusieurs arguments techniques :

1) Notre ALGO répond à tous les critères demandés notamment qu'il se classe parmi les algorithmes d'apprentissage supervisé.
2) Notre ALGO est applicable à des problèmes de Classification et Régression
3) L'ALGO est adapté à l'apprentissage semi-supervisé et plus précisément au self-training. Il dispose de la méthode predict_proba pour la classification et la méthode predict pour la régression (sklearn.linear_model.SGDClassifier — scikit-learn 1.0.2 documentation / sklearn.linear_model.SGDRegressor — scikit-learn 1.0.2 documentation).
4) L'ALGO est potentiellement distribuable (A Brief Introduction to Distributed Training with Gradient Descent | by Shiv Vidhyut | Medium).

- **Motivation personnelle**

Nous sommes familiers avec la logique qu'utilise l'ALGO, c'est la méthode de la descente du gradient que nous l'avons vu en cours d'optimisation. Elle consiste à ajuster les paramètres de façon à minimiser les erreurs du modèle, c'est-à-dire à minimiser la Fonction coût (Log Loss). Pour ça, il faut déterminer comment est-ce-que cette fonction varie en fonction des différents paramètres. C'est pourquoi, on calcule le **Gradient** (dérivée) de la fonction Coût.

Dans le cas du traitement d'une base de données à grande taille, la descente de gradients va calculer toutes les dérivées de la fonction coût en respectant tous les « features » et donc on aura trop d'itérations. C'est ici, que notre ALGO va introduire de l'aléatoire dans l'application de la méthode de la descente du gradient dans le choix des observations de la base de données pour chaque itération afin d'optimiser le temps du calcul.

- **Scénarios pertinents pour l'ALGO**

1) Application du **SGD** sur la base de données « Sonar » accessible sur le dépôt de l'UCI Machine Learning Repository (les variables prédictives sont normalisées et varient entre 0 et 1) afin de reconnaitre le type d'un objet
(Y= 1 = Mine Vs Y = 0 = Rock) ➔ Problème de Classification.

2) Détection automatique de l'effilochage de la chaussée en asphalte à l'aide de l'extraction de caractéristiques basée sur la texture de l'image et de la régression logistique de descente de gradient stochastique. À partir des images des chaussées en asphalte, le SGD est utilisé pour classer l'échantillon d'images en deux catégories (effilochage et non-effilochage).

3) À partir des caractéristiques de différents appartements (superficie, localisation géographique, quartier…), on peut appliquer notre ALGO pour prédire le prix d'un appartement.

4) À partir d'un échantillon des arbres dans un foret dont on a mesuré pour chacun d'eux la taille et le rayon, on peut appliquer notre ALGO pour un nouvel arbre dont on dispose que de la taille pour prédire le rayon ou bien inversement.

Étape 02 : choix de datasets et de paramètres du modèle

- **Choix des jeux de données**

1) Régression

Pour implémenter la régression, on a choisi le jeu de données « Boston » disponible sur la bibliothèque Scikit-Learn. « Boston » est une base de données standard et facile à comprendre qui ne nécessite pas de télécharger de fichiers à partir d'un site Web externe.

Ce jeu de données est adaptable à un problème de régression vu que la variable à prédire est continue (valeur médiane des maisons)

Caractéristiques : 506 observations et 13 attributs(numériques/catégoriales). Prix ($14^{ème}$ attribut) : variable cible.

2) Classification

Quant à la classification, on a utilisé « Iris » vu que ce jeu de données est considéré comme « Hello Word ! » dans le Machine Learning. « Iris » est à juste titre largement utilisé dans la science statistique, en particulier pour illustrer divers problèmes dans les graphiques statistiques, les statistiques multivariées et l'apprentissage automatique.

Caractéristiques : 150 observations et 4 attributs (numériques) + Classe (Iris-Setosa, Iris-Versicolour, Iris-Virginica).

- **Implémentation d'un programme de Test**

Voir le code sur les 2 Notebooks Jupyter (Régression.ipynb et Classification.ipynb)

- **Métriques et résultats obtenus**

1) Régression

Métrique **:** Vu que notre sortie Y est continue, on a choisi la métrique « MSE » (Root Squared Error). L'erreur quadratique moyenne (MSE) est la fonction de perte la plus couramment utilisée pour la régression. Si un vecteur de n prédictions est généré à partir de n observations sur toutes les variables, Y est le vecteur des valeurs observées de la variable prédite et $\hat{Y}$ étant les valeurs prédites. Alors $MSE = \frac{1}{n}\sum_1^n (Y_i - \hat{Y}_i)^2$

Résultats obtenus :

Pour l'implémentation de **SGDRegressor** par Scikit-Learn avec des paramètres par défaut, par exemple **nbre.max d'itérations** = 1000 et critère d'arrêt **tol** = 10^{-3}, on a trouvé une erreur MSE = 0.027. Nous déduisons que notre algorithme est très efficace sur Scikit-Learn.

NB : on a choisi un découpage (70%-30%) pour les bases Train/Test

2) Classification

Notre sortie Y est discrète (3 classes). Après l'implémentation de **SGDClassifier,** on a pensé à utiliser la courbe de ROC mais cette métrique est applicable que pour des problèmes de classification binaire. On est passé à deux autres métriques applicables à des problèmes de classification multi-classes qui sont **la matrice de confusion** et **le score de précision**.

NB : pour un découpage 70%-30% sur Train/Test, on n'a pas obtenu des bons résultats réels et notre algorithme risque d'avoir un sur-apprentissage. Alors on a décidé de prendre un découpage 50%-50% pour augmenter le nombre d'observations du Test.

- **Matrice de confusion**

La matrice de confusion est un outil de mesure de la performance des modèles de classification à 2 classes ou plus. Dans le cas binaire (i.e. à deux classes, le cas le plus simple), la matrice de confusion est un tableau à 4 valeurs représentant les différentes combinaisons de valeurs réelles et valeurs prédites comme dans la figure ci-dessous.

		Reality	
	Confusion matrix	**Negative : 0**	**Positive : 1**
Prediction	**Negative : 0**	**True Negative : TN**	**False Negative : FN**
	Positive : 1	**False Positive : FP**	**True Positive : TP**

Cette matrice est indispensable pour définir les différentes métriques de classification telles que l'Accuracy (score de précision), le F1-score ou encore l'AUC PR et l'AUC ROC.
Résultats :

$$\begin{pmatrix} 24 & 0 & 0 \\ 0 & 24 & 0 \\ 0 & 2 & 25 \end{pmatrix}$$

- Sur 25 valeurs de la 1ère classe, on a eu 24 bonnes prédictions.
- Pareil pour la 2ème classe.
- Pour la 3ème classe, on a eu une bonne prédiction sur tout l'échantillon (25/25).

- **Score de précision**

La précision correspond au nombre d'observations **correctement** attribuées à la classe i par rapport au nombre total d'observations prédites comme appartenant à la classe i (total predicted positive), $Précision = \frac{T_p}{T_p + F_p}$

Résultats **:** on a eu un score de précision égal à 97.333%.
Il semble que notre modèle est trop puissant en l'appliquant sur des petites bases de données comme « Iris ». Comme indiqué avant, on risque d'avoir un sur-apprentissage si on avait choisi 70% de la base pour l'apprentissage du modèle et 30% pour le tester.

Étape 03 : Étude des paramètres du modèle

On va étudier les différents paramètres de notre ALGO pour la régression et la classification en observant les valeurs des métriques qu'on a choisi au début.

1) Régression

1-1) Explication des paramètres

class `sklearn.linear_model.`**`SGDRegressor`**(*loss='squared_error', *, penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=1000, tol=0.001, shuffle=True, verbose=0, epsilon=0.1, random_state=None, learning_rate='invscaling', eta0=0.01, power_t=0.25, early_stopping=False, validation_fraction=0.1, n_iter_no_change=5, warm_start=False, average=False*).
Tous les paramètres de SGDRegressor sont listés en détail sur le lien de Scikit-Learn (sklearn.linear_model.SGDRegressor — scikit-learn 1.1.0 documentation).
Dans notre étude, on va se focaliser seulement sur l'étude de trois paramètres suivants :

- **max_iter** (entier, valeur par défaut = 1000) : nombre maximal d'itérations sur les données d'entraînement. Cela n'a d'impact que sur le comportement de la méthode d'ajustement, et non sur la méthode « partial_fit ».
- **Tol** (réel, valeur par défaut = 10^{-3}) : Le critère d'arrêt. Si ce n'est pas nul, l'entraînement s'arrêtera quand (loss > best_loss - tol) pendant n_iter_no_change itérations consécutives. La convergence est vérifiée par rapport à la perte d'entraînement ou à la perte de validation en fonction du paramètre early_stopping.
- **Shuffle** (booléen, valeur par défaut = True) : Si les données d'entraînement doivent être mélangées après chaque itération ou non.

1-2) Discussion des variations

On va étudier la variation de la métrique 'MSE' pour différentes valeurs de paramètres
Shuffle = True

	Max_iter		
tol	**100**	**1000**	**10000**
0.1	35.825	1.6579	1.187
10^{-3}	28.959	0.11597	0.04469
10^{-6}	1.67	0.06	0.00165

Pour le paramètre **shuffle** par défaut (True) ;

- En augmentant la valeur de **max_iter**, on remarque une diminution de l'erreur ce qui est tout à fait logique vu que le modèle va mieux apprendre en appliquant plus d'itérations.
- En diminuant la valeur de **tol,** on observe le même phénomène (l'erreur décroit) et cela est dû au fait que le modèle ne va pas s'arrêter assez rapidement (et donc mieux apprendre). En particulier, la valeur de (best_loss – loss) doit être inférieure à 10^{-6} pour la 3ème ligne.

Shuffle = False

	Max_iter		
tol	**100**	**1000**	**10000**
0.1	1605127.92	18.7931	18.7931
10^{-3}	1605127.92	5.4348	0.2044
10^{-6}	1605127.92	5.4348	$22.5\ 10^{-5}$

Pour le paramètre **shuffle** = False ;

- Généralement, on observe le même phénomène précédent (l'erreur décroit à chaque fois qu'on augmente le nombre d'itérations ou on diminue la valeur di critère d'arrêt). La valeur optimale est toujours donnée pour **max_iter** = 10000 et **tol** = 10^{-6}.
- On peut aussi remarquer que notre modèle arrive à un niveau où il ne peut plus apprendre même si on diminue la valeur d'arrêt (**tol**) et donc générer la même erreur (cf. **max_iter** = 100).
- Si on implémente le modèle plusieurs fois, il va toujours nous donner le même résultat car les données ne sont plus mélangées avant chaque itération (shuffle = False).

2) Classification

2.1) Explication des paramètres

class `sklearn.linear_model.`**`SGDClassifier`**(*loss='hinge', *, penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=1000, tol=0.001, shuffle=True, verbose=0, epsilon=0.1, n_jobs=None, random_state=None, learning_rate='optimal', eta0=0.0, power_t=0.5, early_stopping=False, validation_fraction=0.1, n_iter_no_change=5, class_weight=None, warm_start=False, average=False*).

Tous les paramètres de SGDClassifier sont listés en détail sur le lien de Scikit-Learn [(sklearn.linear_model.SGDClassifier — scikit-learn 1.1.0 documentation).]

Pareil pour la classification, on va s'intéresser à étudier la variation de trois paramètres **max_iter**, **tol** et **shuffle**.

2.2) Discussion des variations

On va étudier la variation des métriques « matrice de confusion » et « score de précision » pour différentes valeurs des trois paramètres. Chaque case des tableaux ci-dessous contient la matrice de confusion ainsi que la précision du modèle.

Shuffle = True

	Max_iter		
tol	**100**	**1000**	**10000**
0.1	$\begin{pmatrix}24&0&0\\6&27&0\\0&0&18\end{pmatrix}$ 92%	$\begin{pmatrix}24&0&0\\0&24&0\\0&4&23\end{pmatrix}$ 94.667 %	$\begin{pmatrix}24&0&0\\0&24&0\\0&1&26\end{pmatrix}$ 98.667%
10^{-3}	$\begin{pmatrix}24&0&0\\0&24&0\\0&4&23\end{pmatrix}$ 94.667%	$\begin{pmatrix}24&0&0\\0&24&0\\0&2&25\end{pmatrix}$ 97.333%	$\begin{pmatrix}24&0&0\\0&24&0\\0&1&26\end{pmatrix}$ 98.667%
10^{-6}	$\begin{pmatrix}24&0&0\\0&24&0\\0&3&24\end{pmatrix}$ 96%	$\begin{pmatrix}24&0&0\\0&24&0\\0&1&26\end{pmatrix}$ 98.667%	$\begin{pmatrix}24&0&0\\0&24&0\\0&0&27\end{pmatrix}$ 100%

En termes de précision, le modèle s'améliore à chaque fois qu'on augmente le nombre d'itérations ou bien qu'on diminue la valeur de **tol** pour le critère d'arrêt.

Comme indiqué avant, le modèle semble trop puissant pour un dataset de petite taille comme « Iris » (précision optimale = 100% pour **max_iter** = 10000 et **tol** = 10^{-6}.

Shuffle = False

	Max_iter		
tol	**100**	**1000**	**10000**
0.1	$\begin{pmatrix}24&0&0\\0&24&0\\0&4&23\end{pmatrix}$ 94.667%	$\begin{pmatrix}24&0&0\\0&24&0\\0&4&23\end{pmatrix}$ 94.667 %	$\begin{pmatrix}24&0&0\\0&24&0\\0&4&23\end{pmatrix}$ 94.667%

10^{-3}	$\begin{pmatrix}24 & 0 & 0\\ 0 & 24 & 0\\ 0 & 0 & 27\end{pmatrix}$ 100%	$\begin{pmatrix}24 & 0 & 0\\ 0 & 24 & 0\\ 0 & 0 & 27\end{pmatrix}$ 100%	$\begin{pmatrix}24 & 0 & 0\\ 0 & 24 & 0\\ 0 & 0 & 27\end{pmatrix}$ 100%
10^{-6}	$\begin{pmatrix}24 & 0 & 0\\ 0 & 24 & 0\\ 0 & 0 & 27\end{pmatrix}$ 100%	$\begin{pmatrix}24 & 0 & 0\\ 0 & 24 & 0\\ 0 & 0 & 27\end{pmatrix}$ 100%	$\begin{pmatrix}24 & 0 & 0\\ 0 & 24 & 0\\ 0 & 0 & 27\end{pmatrix}$ 100%

La précision de notre modèle a stagné pour **tol**=0.1 pour une valeur de **max_iter** égale à 100, on ne peut plus l'améliorer en augmentant le nombre d'itérations. Il arrive à se converger à partir d'un petit nombre d'itérations.
Le modèle arrive à 100% de précision pour une valeur de **tol** = 10^{-3} et **max_iter** = 100. Cela peut être dû à la petite taille de la base de données.

Étape 04 : Choix de nouveaux jeux de données

1) Classification

1.1) Choix de la base de données

- Pour une deuxième implémentation de la classification, on a choisi le jeu de données « Chronic Kidney disease ». Il est accessible sur le dépôt de l'UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets/chronic_kidney_disease).

- Pour notre recherche, notre jeu de données devrait être utilisable pour des problèmes de classification, facile à comprendre ses features et sa variable cible ainsi qu'avoir des valeurs manquantes pour pouvoir passer par la phase de nettoyage et prétraitement de données.
  Notre dataset peut être utilisé pour classifier les patients ayant une maladie rénale chronique en se basant sur les antécédents cliniques, les examens physiques et les tests de laboratoire.

1.2) Analyse du dataset

- Notre base de données présente 400 observations de 24 variables explicatives (age, blood pressure, sugar, red cells count, anemia,…) et une variable cible (**class** : 'ckd' ou 'notckd') indiquant si oui ou non le patient présente la maladie rénale.

```
(400, 25)
Index(['age', 'bp', 'sg', 'al', 'su', 'rbc', 'pc', 'pcc', 'ba', 'bgr', 'bu',
       'sc', 'sod', 'pot', 'hemo', 'pcv', 'wbcc', 'rbcc', 'htn', 'dm', 'cad',
       'appet', 'pe', 'ane', 'class'],
      dtype='object')
```

- On a interrogé le dataset pour vérifier s'il contient des valeurs manquantes ou pas.

```
df.isna().sum()
```

```
age          9
bp          12
sg          47
al          46
su          49
rbc        152
pc          65
pcc          4
ba           4
bgr         44
bu          19
sc          17
sod         87
pot         88
hemo        52
pcv         70
wbcc       105
rbcc       130
htn          2
dm           2
cad          2
appet        1
pe           1
ane          1
class        0
dtype: int64
```

1.3) Nettoyage et préparation de données

Pour le nettoyage et le traitement des valeurs manquantes, on a procédé comme suit :

- Division du dataset en 2 partitions, X qui contient les variables explicatives et Y qui contient la variable cible

```
#X contient les données des variables explicatives, Y contient la variable cible (à prédire)
X = df.drop("class", axis =1)
Y = df["class"]
```

- On a séparé les attributs numériques des attributs catégoriels de X pour pouvoir appliquer '**SimpleImputer'** qui va remplacer les valeurs manquantes numériques par la médiane et les valeurs catégorielles par les valeurs les plus fréquentes.

Valeurs numériques :

```
# Replace missing values with a median
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median")
#Create a copy without text attribute
#Nominal attributes:"rbc","pc","pcc","ba","htn","dm", "cad", "appet", "pe", "ane"
X_num = X[["age", "bp","sg", "al", "su", "bgr", "bu", "sc", "sod", "pot", "hemo","pcv", "wbcc", "rbcc"]]
```

```
#fit the imputer instance to the data
imputer.fit(X_num)
#imputer.statistics_
X_num.median().values
```

```
#trasform the dataset by replacing the missing values by the medians
I = imputer.transform(X_num)
X_num_tr = pd.DataFrame(I, columns = X_num.columns)
```

Valeurs catégorielles :

```
X_cat = X[["rbc","pc","pcc","ba","htn","dm", "cad", "appet", "pe", "ane"]]

#Use the pandas functions : factorize(), Imputer with the "most_frequent" strategy to preprocess and
#transform the categorical attributes

X_cat_encoded = []
X_categories = []
for x in X_cat:
    X_cat_encod, X_categ = pd.factorize(X_cat[x])
    X_cat_encoded.append(X_cat_encod)
    X_categories.append(X_categ)
print(X_cat_encoded)
print(X_categories)
```

```
#replace missing categorical values by the most frequent ones
X_cat_encoded = pd.DataFrame(np.transpose(X_cat_encoded))
X_cat_encoded.info()
imputer = SimpleImputer(missing_values= -1, strategy="most_frequent")
#fit the imputer instance to the data
imputer.fit(X_cat_encoded)
imputer.statistics_
# trasform the dataset by replacing the missing values by the most frequent values
X_cat_enc = imputer.transform(X_cat_encoded)
X_cat_tr = pd.DataFrame(X_cat_enc, columns = X_cat.columns)
```

À la fin, on a fusionné les 2 bases traitées en une base finale X_tr.

```
# Merge the numerical and categorical datasets
X_tr = pd.concat([X_num_tr, X_cat_tr], axis=1)
```

- Pour la variable cible Y, On a remplacé les valeurs 'ckd' par 1 et 'notckd' par 0.

```
Y[Y=="ckd"]=1
Y[Y=="notckd"]=0
```

- Découpage du dataset en Train/Test (50%-50%) avec un choix aléatoire des observations afin d'éviter un surapprentissage.

```
# split the data set into train and test
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.3, random_state=1)
```

- On a utilisé la fonction **'Standard Scalar'** pour la standardisation des données, on l'a appliquée seulement sur les données d'apprentissage

```
# Import the library
from sklearn.preprocessing import StandardScaler

# Use StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
scaled_data = scaler.transform(X_train)
```

```
X_train = pd.DataFrame(data = X_train, columns=X_tr.columns)
X_train['class'] = list(y_train)
X_test = pd.DataFrame(data = X_test, columns=X_tr.columns)
X_test['class'] = list(y_test)
```

2) Régression

2.1) Choix de la base de données

- Pour implémenter la régression, on est parti sur le dataset « Combined Cycle Power Plant » disponible sur le dépôt de l'UCI Machine Learning Repository (UCI Machine Learning Repository: Combined Cycle Power Plant Data Set).
- Ce jeu de données est utilisé pour des problèmes de régression vu qu'il présente une variable cible continue (la production horaire nette d'énergie électrique), il est facile à comprendre avec seulement 4 variables explicatives. En plus, il présente 9568 observations donc notre modèle va apprendre sur une échelle un peu plus grande.
- L'ensemble de données est recueilli à partir d'une centrale à cycle combiné sur 6 ans (2006-2011), lorsque la centrale a été mise en service à pleine charge.

2.2) Analyse du dataset

Notre base de données présente 9568 observations de 4 variables explicatives (Temperature, Ambient pressure, Relative Humidity et Exhaust Vacuum) et une variable cible continue (EP) indiquant la production horaire nette d'énergie électrique.

```
Index(['AT', 'V', 'AP', 'RH', 'PE'], dtype='object')
```

- Le dataset choisi ne présente aucune valeur manquante donc, nous sommes dispensés de la phase de nettoyage de données.

```
df.isna().sum()
```

```
AT    0
V     0
AP    0
RH    0
PE    0
dtype: int64
```

2.3) Préparation de données

Pour la préparation de données, on a procédé comme suit :

- Division du dataset en 2 partitions, X qui contient les variables explicatives et Y qui contient la variable cible

```
X = df.drop("PE", axis =1)
Y = df["PE"]
X=X.to_numpy()
Y=Y.to_numpy()
```

- Découpage du dataset en Train/Test (70%-30%) avec un choix aléatoire des observations

```
# split the data set into train and test
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.3, random_state=1)
```

- On a utilisé la fonction '**Standard Scalar'** pour la standardisation des données, on l'a appliquée seulement sur les données d'apprentissage

```
scaler = preprocessing.StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

```
X_train = pd.DataFrame(data = X_train, columns = [["AT","V","AP","RH"]])
X_train['PE'] = list(y_train)
X_test = pd.DataFrame(data = X_test, columns = [["AT","V","AP","RH"]])
X_test['PE'] = list(y_test)
```

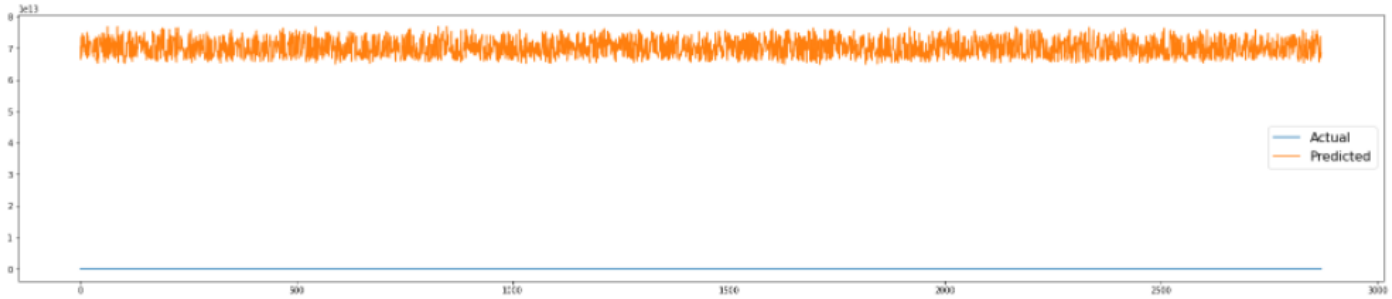
Étape 05 : Test du modèle sur les nouvelles bases de données

Dans cette étape, on va réimplémenter notre modèle pour la régression et la classification sur nos propres bases de données choisies dans l'étape précédente « chronical kidney disease » et « Combined Cycle Power Plant » après avoir nettoyé et prétraité les données. Cela est fait après calibration de paramètres pour obtenir de meilleurs résultats.

1) Régression

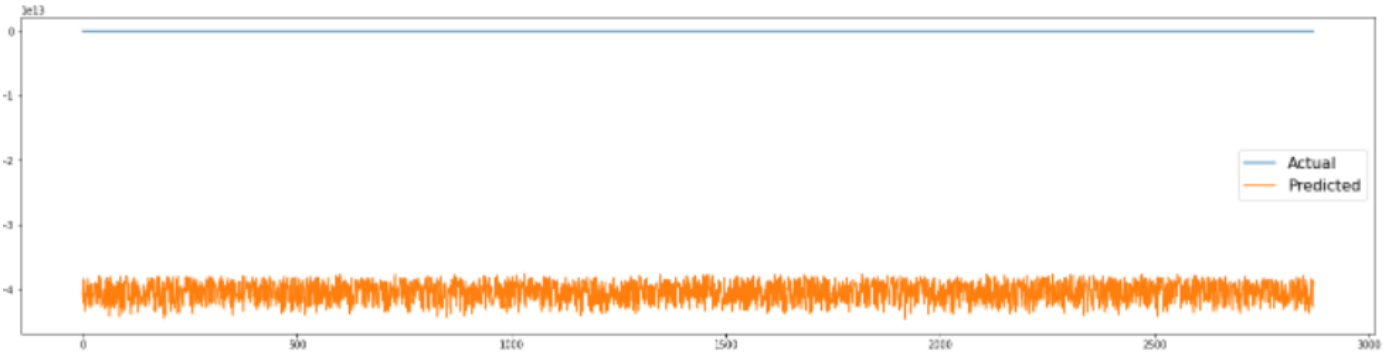
1.1) Implémentation du modèle de régression avec des paramètres par défaut

Dans un premier lieu, on est parti sur l'implémentation du modèle par des paramètres par défaut.
On a eu comme résultat une prédiction y_pred (à l'ordre de 6 10^{13}) trop éloignée par rapport à la sortie actuelle y_test(à l'ordre de 454), ce qui nous a donné une erreur quadratique trop grande (4. 10^{27}) et inacceptable en termes de précision. Notre modèle semble qu'il n'a rien appris en phase d'apprentissage vu qu'il nous a généré une fausse prédiction.



1.2) Ajustement des paramètres choisis à l'étape 3 (max_iter, tol et shuffle)

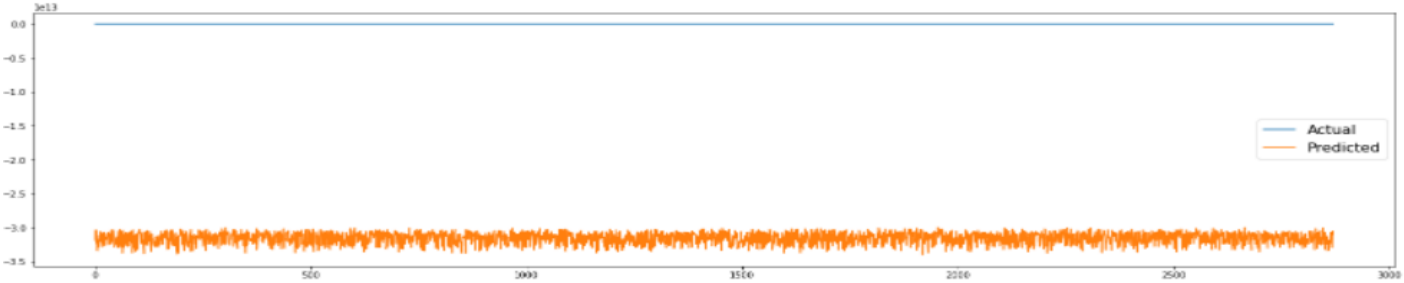
Notre première réflexion était de réajuster les paramètres étudiés dans la troisième étape tout en espérant avoir une bonne prédiction. On va garder le paramètre **shuffle** = True qui sert à mélanger les données à chaque itération, on est parti sur **max_iter** = 10000 et **tol** = 10^{-6}, ces deux derniers nous ont donné la meilleure prédiction dans la troisième étape.



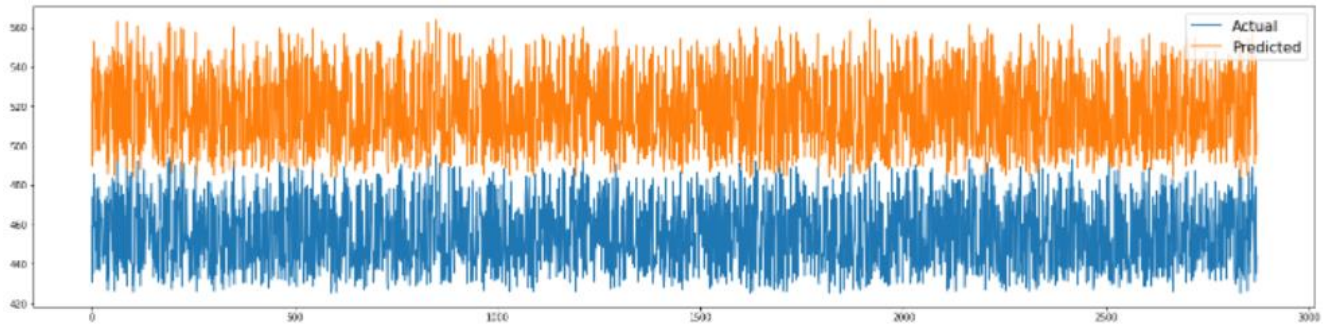
Le problème persiste, notre prédiction est encore fausse et l'erreur est toujours énorme (1.63 10^{27}).
On déduit que le problème n'est pas lié à ces trois paramètres et il fallait qu'on passe à la calibration d'autres paramètres du modèle.
➔ Après recherche et analyse de différents paramètres, on a conclu que le problème est lié au type de la fonction de perte (**loss**)**.** Le SGD Regressor utilise par défaut une fonction de perte **loss** = 'squared_error'.

1.3) Ajustement de la fonction de perte

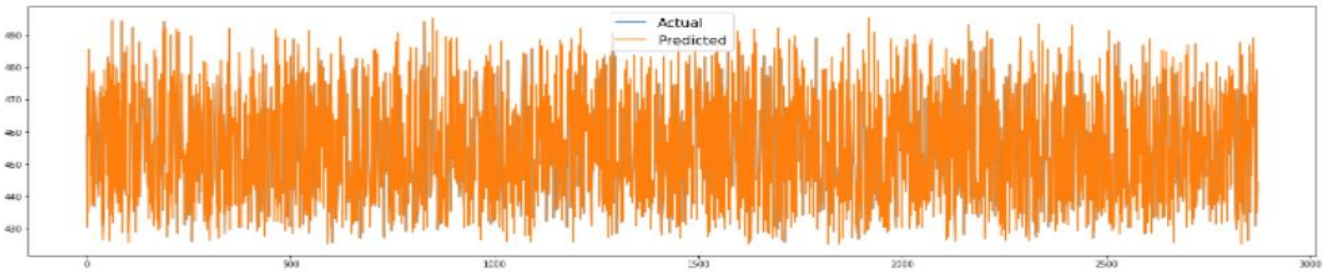
On a réimplémenté le modèle avec les autres fonctions de pertes possibles pour le modèle et on est arrivé à avoir de bonne prédiction comme présenté ci-dessous :
Fonction de perte = 'squared_epsilon_insensitive'



Fonction de perte = 'epsilon_insensitive'



Fonction de perte = 'huber'



On a résumé les erreurs fournies dans le tableau suivant :

loss	***squared_error***	***squared_epsilon_insensitive***	***epsilon_insensitive***	***huber***
MSE	$1.63\ 10^{27}$	$9.97\ 10^{26}$	3965.25	0.02

Comme indiqué dans le tableau, la plus petite erreur (acceptable) est générée par un modèle qui utilise la fonction de perte de type 'huber'. Avec **max_iter** = 10000, **tol** = 10^{-6} et **shuffle** = True, on a obtenu une très bonne prédiction avec une erreur très petite égale à 0.02.
En effet, la fonction de perte '*huber*' est une fonction de perte utilisée dans la régression robuste, qui est moins sensible aux valeurs aberrantes dans les données que la perte '*squared_error*'.

2) Classification

2.1) Implémentation du modèle de classification avec des paramètres par défaut

On a eu une bonne prédiction avec un score de précision très élevé qui est égal à 99.5% et avec la matrice de confusion suivante :

$$\begin{pmatrix} 75 & 0 \\ 1 & 124 \end{pmatrix}$$

2.2) Implémentation du modèle avec d'autres valeurs de paramètres

On a étudié la variation de la précision du modèle tout en manipulant les valeurs de **max_iter** et **tol** et en gardant **shuffle** = True.

max_iter = 100, tol = 0.1

On a eu une précision plus petite mais toujours forte (99 %), avec la matrice de confusion suivante :

$$\begin{pmatrix} 75 & 0 \\ 2 & 123 \end{pmatrix}$$

max_iter = 10000, tol = 10^{-6}

On est arrivé à une précision optimale (100 %), le modèle est trop efficace par rapport à une base de test de 200 observations. La matrice de confusion est la suivante :

$$\begin{pmatrix} 75 & 0 \\ 0 & 125 \end{pmatrix}$$

Notre modèle de classification arrive à classifier les patients malades des patients seins en le faisant apprendre sur une base d'apprentissage de seulement 200 observations.
Le SGD Classifier semble trop performant pour des bases de données de petite taille.
On peut voir maintenant pourquoi la descente de gradient stochastique est si populaire.

Étape 06 : Apprentissage semi-supervisé

On a réimplémenté le modèle de classification SGDClassifier en apprentissage semi-supervisé sur le dataset «Iris ». L'apprentissage semi-supervisé consiste à diviser notre base d'apprentissage **X_train** en 2 partitions **Lab** et **nonLab,** on va entrainer le modèle sur Lab et l'améliorer sur nonLab. Enfin, on va le tester sur la base **X_test**.

1) Construction d'une fonction maskData

Vu que notre dataset est composé des données étiquetées, on a commencé par mettre en place une fonction qui prend comme paramètre **y_train** et qui nous rend une base non labelisée à un pourcentage donné. Ce dernier va varier ensuite entre 10% et 80%.

```
import random

def maskData(true_labels, percentage):

    mask = np.ones((1,len(true_labels)),dtype=bool)[0]
    labels = true_labels.copy()

    for l, enc in zip(np.unique(true_labels),range(0,len(np.unique(true_labels)))):

        deck = np.argwhere(true_labels == l).flatten()
        random.shuffle(deck)

        mask[deck[:int(percentage * len(true_labels[true_labels == l]))]] = False

        labels[labels == l] = enc

    labels[mask] = -1

    return np.array(labels).astype(int)
```

2) Implémentation du modèle sur une base Lab de 10%

```
from sklearn import datasets
from sklearn.semi_supervised import LabelPropagation

def runLP():

    #UNLABEL 90% OF THE DATASET
    masked_labels = maskData(y_train, 0.1)

    #RUN THE MODEL
    model = SGDClassifier(max_iter = 10000, tol =1e-6)
    model.fit(X_train, masked_labels)
    y_pred = model.predict(X_test)

    #PRINT CONFUSION MATRIX
    print(confusion_matrix(y_test, y_pred))
    print(round(accuracy_score(y_test,y_pred)*100,3),'%')


if __name__ == '__main__':
    runLP()
```

3) Analyse des résultats

- Comme demandé, on a gardé une base test de 10% de la base initiale et on a divisé le reste en Lab/nonLab.
- Après avoir varié la taille des bases Lab/nonLab, on a observé une amélioration au niveau du score de précision du modèle. À chaque fois qu'on augmente la proportion du Lab (données étiquetées), le modèle s'améliore et arrive à générer une prédiction avec une meilleure précision.
- Le graphe suivant nous montre l'amélioration du modèle en termes de précision en fonction de la taille de la base Lab.
- On peut conclure que la taille de données étiquetées peut intervenir sur la performance d'un modèle d'apprentissage automatique.