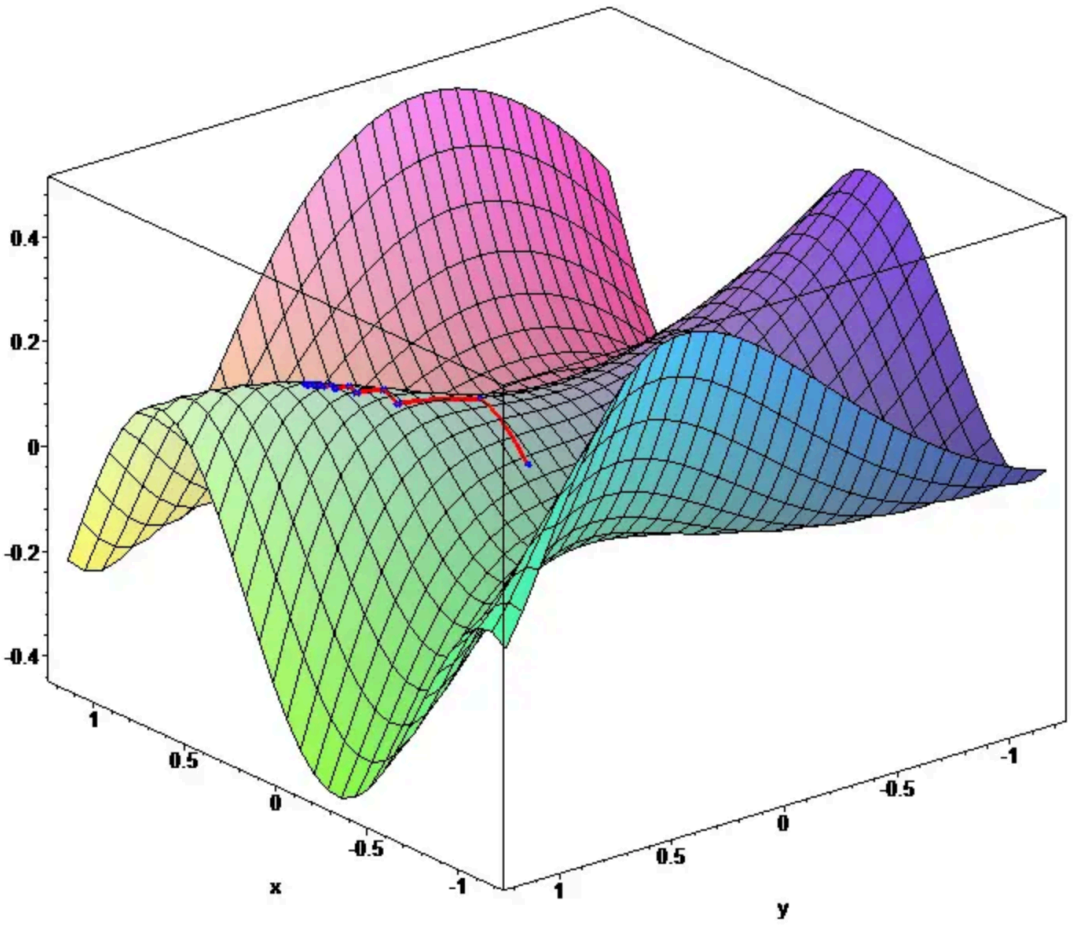
regressions linéaires et machine learning

Lancelot Ravier, Martin Combelles



Introduction

Ce projet est une présentation finale issue de notre participation au data challenge *Chronological Age Prediction Challenge 1.1 (epiclock1.1)* visant à prédire l'age d'un groupe d'individus en fonction de leur profile de methylation d'ADN sanguin. L'objectif de ce challenge était d'entrainer des modèles statistiques ou de machine learning sur les données de methylation. Nous avons choisi certains modèles que nous allons vous présenter ci-dessous.

1) Méthode

1.1) Base de données

1.1.1) Modification de l'extension des bases de données

Les bases de données issues du data challenge ont été créées au format .RDS (bases de données R). Afin de permettre une analyse des données dans d'autres langages de programmation (dans notre cas : Python), nous avons importé les bases de données data_train.RDS et data_test.RDS sous R, puis nous les avons exportées au format .csv, en executant le code suivant :

```
# Définition de l'espace de travail
setwd("~/Documents/GitHub/data_challenge_epiclock/projet")

# Importation des bases de données au format .rds
data_train <- readRDS("data_train.rds")
data_test <- readRDS("data_test.rds")

# Exportation des bases de données au format .csv
write.csv(data_train, "data_train.csv")
write.csv(data_test, "data_test.csv")
```

1.1.2) Importation et nettoyage des données

Afin de supprimer le risque d'erreurs lors de l'execution des fonctions, nous devons traiter les valeurs manquantes et supprimer les colones inutiles générées par la modification de l'extension initiale des bases de donneés fournies. Pour ce faire, nous utilisons les fonctions du package *Pandas (Python)* :

```
import pandas as pd
import numpy as np

# Importation des bases de données
data_train = pd.read_csv("data_train.csv")
data_test = pd.read_csv("data_test.csv")
```

```
# Verification de la présence de NaN
data_train.isnull().values.any()

# Suppression des colonnes inutiles
data_cleaned = data_train.drop(columns=["Unnamed: 0", "gender"])
data_cleaned = data_cleaned.dropna()
```

1.1.3) Découpage et normalisation des données

```
from sklearn.model_selection import train_test_split

# Séparation des variables explicaives et de la variable expliquée
X = data_cleaned.drop(columns=["age"])
y = data_cleaned["age"]

# Création des partitions train et test (utilisation d'une seed pour la reproductibilité)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=42)
```

Afin de pouvoir comparer les différents modèles de machine learning, nous avos séparé la base de données en deux partitions : - La partition d'entrainement (90%) sur laquelle nos différents modèles seront entrainés - La partition de test (10%) dont la variable "age" sera prédite par nos différents modèles à des fins de comparaisons en pré-soumission.

Cette découpe de la base de données est primordiale pour analyser les modèles entre-eux et détécter certains résultats abérants ou erreurs de code. En effet, la séparation train/test permet d'entrainer et d'évaluer nos modèles sur la même base de données et de comparer les résultats en amont d'une soumission sur la plateforme de data challenge, offrant souvent un nombre limité de soumissions. De plus, cette vérification permet d'ajuster les hyperparamètres des modèles afin d'affiner leur précision à l'aide de la métrique d'évaluation choisie.

```
from sklearn.preprocessing import StandardScaler

# Normalisation des données
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Ensuite, les données ont été normalisées afin de pouvoir entrainer les modèles sur des données sans unité et éviter que des modèles à pénalité comme Ridge ou Lasso ne portent trop d'importance à des variables aux valeurs disproportionnées lors de la phase d'entrainement. Aussi, certains modèles comme les SVM, les KNN, et les réseaux de neurones calculent des distances ou des relations entre les points (ex : produits scalaires). Si les données ne sont pas

normalisées, les dimensions avec des valeurs plus grandes peuvent bienser les distances ou les pondérations, diminuant ainsi fortement la qualité de notre modèle.

1.2) Modèles

```
from xgboost import XGBRegressor
from sklearn.linear_model import LinearRegression, RidgeCV, LassoCV, ElasticNetCV
```

1.2.1) Choix des différents modèles

Afin d'explorer au mieux les relations entre les variables de notre base de données, nous avons fait le choix de commencer notre analyse par la comparaison de deux modèles distincts aux spécificités différentes :

- Le modèle XGBoost (Extreme gradient Boosting), basé sur des arbres de décision.
- Le modèle linéaire simple (package sklearn - linear_model)

Ensuite, nous avons choisi de comparer les modèles à pénalité de Ridge et de Lasso afin d'analyser les avantages et limites de ces deux modèles appliqués à nos données.

Enfin, nous avons choisi d'entrainer un modèle ElasticNet, combinant les pénalités de Ridge et de Lasso.

1.2.2) Métrique de comparaison des différents modèles

La métrique d'évaluation des soumissions choisie dans le cadre de ce data challenge est le RMSE (Racine carrée de l'erreur quadratique moyenne) :

$$RMSE = \sqrt{\sum_{i=1}^{n} \frac{(\hat{y}_i - y_i)^2}{n}}$$

Sous Python, nous calculons le temps d'entrainement des modèles à l'aide du package time ainsi que le RMSE pour chaque modèle à l'aide du package sklearn.metrics :

```
import time
from sklearn.metrics import root_mean_squared_error

def rmse(y_test, y_pred):
  return root_mean_squared_error(y_test, y_pred)
```

Enfin, nous avons choisi de soumettre nos résultats, non pas sous forme de modèles linéaires avec noms de variables, mais sous forme de prédictions de nos modèles à partir de la base de données *data_test* car certains de nos modèles sont non-paramétriques et ne permettent donc pas de retourner un modèle directement interprétable avec noms de variables (cf. XGBoost). nous effectuerons donc les prédictions à partir de la base data-test en amont de la soumission.

2) Résultats

2.1) Comparaison entre XGBoost et LinearModel

```
start_time = time.time()

XGBRegressor_model = XGBRegressor()
XGBRegressor_model.fit(X_train_scaled, y_train)
print("Temps d'entrainement du modèle : %s seconds" % (time.time() - start_time))

y_pred_xgboost = XGBRegressor_model.predict(X_test_scaled)
rmse_xgboost = rmse(y_test, y_pred_xgboost)

print("XGBoost RMSE = ",rmse_xgboost)
```

```
Temps d'entrainement du modèle : 23.297702074050903 seconds
XGBoost RMSE =  7.042929651378303
```

```
start_time = time.time()

LinearRegression_model = LinearRegression()
LinearRegression_model.fit(X_train_scaled, y_train)
print("Temps d'entrainement du modèle : %s seconds" % (time.time() - start_time))

y_pred_lm = LinearRegression_model.predict(X_test_scaled)
rmse_lm = rmse(y_test, y_pred_lm)

print("LinearRegression RMSE = ",rmse_lm)
```

```
Temps d'entrainement du modèle : 0.5064661502838135 seconds
LinearRegression RMSE =  3.1801758826480753
```

- Le modèle XGBoost d'arbres de décisions entrainé sur notre base de données de test retourne un RMSE de 7.0429. L'entrainement du modèle à duré 24 secondes.
- Le modèle de régression linéaire simple retourne un RMSE à 3.1802. L'entrainement du modèle a duré une demi seconde.
- L'écart d'erreur moyenne entre les deux modèles est de 3.86

2.2) Comparaison entre RidgeCV et LassoCV

```
start_time = time.time()

RidgeCV_model = RidgeCV()
RidgeCV_model.fit(X_train_scaled, y_train)
print("Temps d'entrainement du modèle : %s seconds" % (time.time() - start_time))

y_pred_RidgeCV_model = RidgeCV_model.predict(X_test_scaled)

rmse_RidgeCV_model = rmse(y_test, y_pred_RidgeCV_model)
print("Ridge RMSE = ",rmse_RidgeCV_model)
print("Nombre de coefficients du modèle : ", (RidgeCV_model.coef_ != 0).sum())
```

```
Temps d'entrainement du modèle : 0.5313262939453125 seconds
Ridge RMSE =  3.1807055843844014
Nombre de coefficients du modèle :  10000
```

```
start_time = time.time()

LassoCV_model = LassoCV()
LassoCV_model.fit(X_train_scaled, y_train)
print("Temps d'entrainement du modèle : %s seconds" % (time.time() - start_time))

y_pred_LassoCV_model = LassoCV_model.predict(X_test_scaled)

rmse_LassoCV_model = rmse(y_test, y_pred_LassoCV_model)
print("Lasso RMSE = ",rmse_LassoCV_model)
print("Nombre de coefficients du modèle : ", (LassoCV_model.coef_ != 0).sum())
```

```
Temps d'entrainement du modèle : 33.45849299430847 seconds
Lasso RMSE =  3.29864410399879
Nombre de coefficients du modèle :  191
```

- L'erreur moyenne du modèle de Ridge avec hyperparamètres optimaux est de 3.1807 avec un temps d'éxécution de moins d'une seconde. Le modèle contient 10000 coefficients $\neq 0$.
- L'erreur moyenne du modèle de Lasso avec hyperparamètres optimaux est de 3.2986. Le modèle contient 191 coefficients $\neq 0$.
- L'écart d'erreur moyenne entre les deux modèles est de 0.11.

2.3) ElasticNet

```
start_time = time.time()
# Initialisation du modèle
ElasticNetCV_model = ElasticNetCV()
ElasticNetCV_model.fit(X_train_scaled, y_train)
print("Temps d'entrainement du modèle : %s seconds" % (time.time() - start_time))

y_pred_ElasticNetCV_model = ElasticNetCV_model.predict(X_test_scaled)
rmse_ElasticNetCV_model = rmse(y_test, y_pred_ElasticNetCV_model)

print("ElasticNet RMSE = ", rmse_ElasticNetCV_model)
print("Nombre de coefficients du modèle : ", (ElasticNetCV_model.coef_ != 0).sum())
```

```
Temps d'entrainement du modèle : 37.338955879211426 seconds
ElasticNet RMSE =  3.1495400988501907
Nombre de coefficients du modèle :  356
```

- Le modèle ElasticNet avec recherche d'hyperparamètres optimaux a un RMSE de 3.1495 sur les donneés de test locales, avec un temps d'execution de 39 secondes et 356 coefficients $\neq 0$ retenus.

3) Discussion

Annotation : l'entrainement et les prédiction des modèles décrits ci-dessus ont été effectuées sur la mème base de données normalisée. Anisi, la métrique calculée (ici, le RMSE) est comparable entre tous nos modèles. Cependant, les résultats soumis sur Codabench ont été prédits à partir de modèles entrainés sur la totalité de la base de données data_train.csv, permettant ainsi des prédictions théoriquement plus précises car les modèles sont entrainés sur une base de données 10% plus grande.

3.1) Comparaison entre les différents modèles

3.1.1) XGBoost v.s. LinearRegression

Lors de la présentation des différents modèles, le modèle XGBoost a été introduit comme un modèle permettant de capter des relations complèxes entre les variables, là où le modèle LinearRegression suppose une relation exclusivement linéaire entre les variables. De ce fait, L'intuition derrière ce choix de comparaison réside dans le fait que la différence de RMSE entre les deux modèles devrait ètre flagrante et permettre une interprétation rapide de la structure de nos données, ainsi que des relations entre les différentes variables. Dans le cas de relations lineaires, le modèle de régression linéaire simple serait plus adapté face au modèle XGBoost car ce dernier tentera de modéliser des relations inexistante pour nos données et ses prédictions sur les données de test seront moins précises car les relations non linéaires que le modèle prédiera ne seront pas valables dans le cas de notre base de données. De plus, cette surcharge de

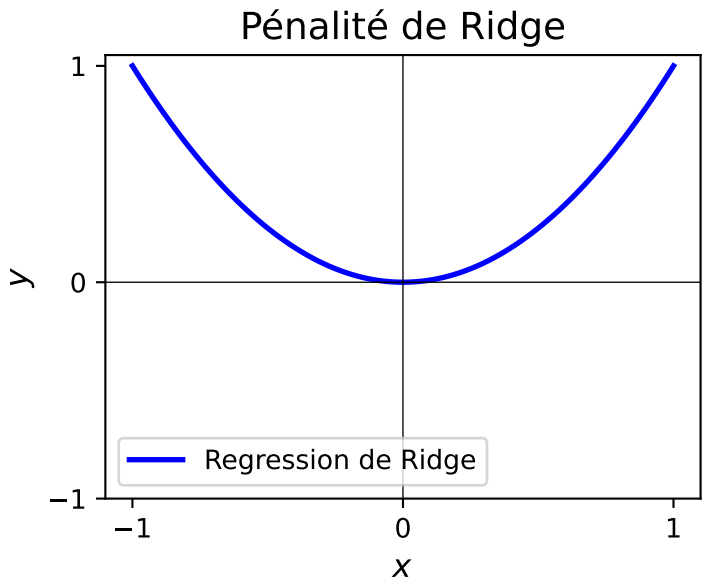
calcul apportée par la complexité des relations modélisées par XGBoost ralentiront le temps d'execution de la phase d'entrainement face au modèle linéaire simple. A partir des résultats présentés en 2.1) , l'écart de RMSE entre nos deux modèles est significatif : en moyenne, les erreurs de prédictions du modèle XGBoost sont deux fois plus importantes que celles du modèle de régression linéaire simple. Ainsi, l'intuition derrière cet écatt important est qu'il est très probable que les relations entre nos données soient linéaires. Ainsi, ce résultat pousse notre choix vers des modèles de regression linéaire.

3.1.2) Ridge v.s. Lasso

La régression de **Ridge** a pour objectif principal de réduire la multicolinéarité et la compléxité du modèle initial sans selection de variables. La fonction que le modèle de Ridge cherche à minimiser est la suivante :

$$\arg\min_{\beta\in\mathbb{R}^p} \underbrace{\frac{1}{2n}||y-X\beta||_2^2}_{Regression} + \underbrace{\lambda\sum_{j=1}^{p}\beta_j^2}_{PenaliteL_2}$$

```python
import numpy as np
import matplotlib.pyplot as plt

# Création des valeurs pour x
x = np.linspace(-1, 1, 100)

# Fonction pour la pénalité Ridge (norme L2) : y = x^2
ridge = x**2

# Création du graphique
plt.figure(figsize=(4, 3))
plt.plot(x, ridge, color='blue', label='Regression de Ridge', linewidth=2)
plt.axhline(0, color='black',linewidth=0.5)  # Axe horizontal
plt.axvline(0, color='black',linewidth=0.5)  # Axe vertical
plt.xticks([-1, 0, 1])
plt.yticks([-1, 0, 1])
plt.xlabel('$x$', fontsize=12)
plt.ylabel('$y$', fontsize=12)
plt.title("Pénalité de Ridge", fontsize=14)
plt.legend()

plt.grid(False)
plt.show()
```



La pénalité Ridge utilise la norme L_2, représentée ici par la courbe quadratique bleue $y = x^2$. Cette pénalité ajoute une contrainte sur la somme des carrés des coefficients. Ainsi, la pénalité sera d'autant plus élevée que les coefficients sont grands. Comme nous pouvons le voir sur le graphique, la pénalité de Ridge a comme effet la réduction des coefficients, sans jamais les annuler completement. Cette pénalité aide à gérer la multicolinéarité et permet de gérer le risque de surajustement en réduisant la variance du modèle. Selon la courbe de Ridge, les coefficeints sont progressivement réduits à mesure qu'ils s'éloignent de 0.

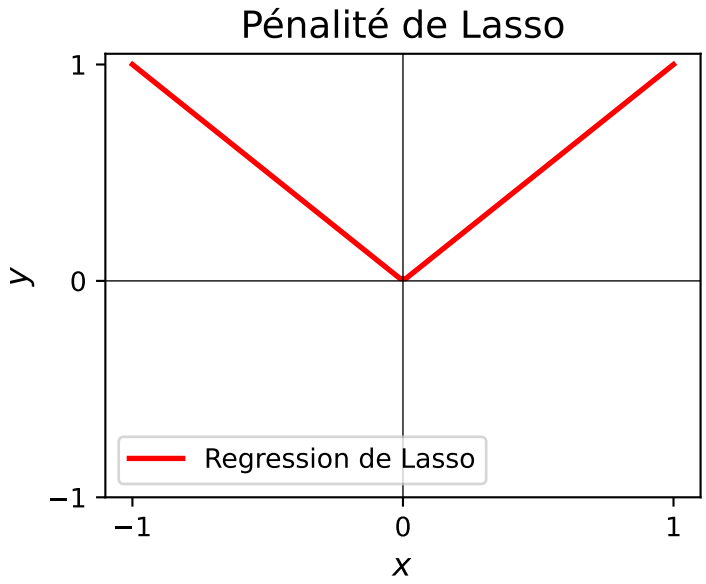
La fonction RidgeCV va permettre d'ajuster, par itération, la froce de régularisation afin de réduire les coefficients de la manièr la plus uniforme possible afin de réduire la colinéarité au maximum.

La regression de **Lasso** a pour objectif la selection des variables les plus pertinentes. La régression de Lasso cherche à minimiser la fonction suivante :

$$\arg\min_{\beta \in \mathbb{R}^p} \underbrace{\frac{1}{2n}||y - X\beta||_2^2}_{Regression} + \underbrace{\lambda \sum_{j=1}^{p} |\beta_j|}_{PenaliteL_1}$$

```
x = np.linspace(-1, 1, 100)

# Fonction pour la pénalité Lasso (norme L1) : y = |x|
lasso = np.abs(x)

# Création du graphique
```

```
plt.figure(figsize=(4, 3))
plt.plot(x, lasso, color='red', label='Regression de Lasso', linewidth=2)

plt.axhline(0, color='black',linewidth=0.5)  # Axe horizontal
plt.axvline(0, color='black',linewidth=0.5)  # Axe vertical
plt.xticks([-1, 0, 1])
plt.yticks([-1, 0, 1])
plt.xlabel('$x$', fontsize=12)
plt.ylabel('$y$', fontsize=12)
plt.title("Pénalité de Lasso", fontsize=14)

plt.legend()
plt.grid(False)
plt.show()
```



La pénalité Lasso utilise la norme L_1, représentée dans le graphique par la fonction $y = |x|$ tracée ici en rouge. Cette pénalité impose une contrainte sur la somme des valeurs absolues des coefficients. Cette contrainte va forcer certains coefficients à devenir exactement nuls, permettant ainsi une selection automatique des variables. Le point angulaire de la courbe en rouge montre que lorsque la valeur du coefficient est très proche de 0, la pénalité de Lasso tends à l'annuler. Cette pénalité est donc très utile dans le cas de données avec un grand nombre de variables explicatives car la regression de Lasso va permettre une selection automatique de variables les plus significatives.

Ici, fonction de recherche d'hyperparamètres optimisée par cross-validation va ajuster la force

de la pénalité de séléction : plus λ sera grand, plus le nombre de variables éliminées va augmenter.

En comparant le RMSE entre les régressions de Ridge et de Lasso, nous remarquons que Ridge donne de meilleurs résultats sur la base de données sur laquelle il a été entrainé. Cependant, lorsque l'on regarde le nombre de paramètres du modèle, on remarque que Ridge a séléctionné tous les paramètres du modèle, alors que Lasso en a séléctionné seulement 191. Ce choix de séléction illustr ebien le caractère plus permissif de Ridge qui ne cherche pas à faire tomber les paramètres à 0, mais à réduire l'influence de l'ensemble des paramètres en les poussant vers 0, sans atteindre ce chiffre. Ainsi, la multicolinéarité et la compléxité du modèle ets réduite. La régression de Lasso à quant à elle gardé les coefficients les plus significatifs, et a annulé la plupart des autres coefficients avec comme effet un nombre de coefficients réduit. Seulement, ce groupe réduit de coefficients sont en réalité les plus significatifs car non touchés par la pénalité de Lassso. Dans notre cas, nous avons un nombre conséquent de variables pur la quasi-totalité proches de 0. Ainsi, il ne serait pas logique de comparer les performances de nos modèles Ridge et Lasso car le modèle de Ridge donnera un RMSE toujours inférieur à celui de ridge car, avec un nombre de variables explicatives bien supérieur au nombre d'observations, le modèle de Ridge peut s'ajuster excessivement bien aux données d'entrainement. Cependant, malgré des performances meilleures en entrainement avec validation croisée en local, la généralisation du modèle de Ridge aux données de test non connues sera extremmement mauvaise (RMSE sur Codabench pour la régression de Ridge : 47).

Ainsi, la régression de Lasso surpassera celle de Ridge lors de la soumission des résultats car le modèle de Lasso sélectionne les variables réellement pertinentes et évite les effets de bruit ou de corrélation excessive, ce qui favorise une meilleure généralisation.

3.1.3) Régression ElasticNet

La régression ElasticNet est une régression linéaire qui a pour particularité de combiner les pénalités de Ride (L_2) et de lasso (L_1). Le problème d'optimisation de la régression ElasticNet est le suivant :

$$\min_{\beta} \frac{1}{2n}||y - X\beta||_2^2 + \lambda_1 \sum_{j=1}^{p} |\beta_j| + \frac{\lambda_2}{2} \sum_{j=1}^{p} \beta_j^2$$

La régression ElasticNet va donc minimiser l'erreur quadratique moyenne ($||y - X\beta||_2^2$) en ajoutant les pénalités de Lasso ($\lambda_1 \sum_{j=1}^{p} |$) encourageant la séléction de variable, puis de Ridge ($\frac{\lambda_2}{2} \sum_{j=1}^{p} \beta_j^2$) régularisant les coefficients pour réduire l'impact de la multicolinéarité. L'avantage de la régression ElasticNet face à la régression de Lasso est que, contraitement à cette dernière ne pouvant séléctionner qu'une seule variable dans un groupe de variable corrélées, ElasticNet va répartir les coefficients sur plusieurs variable corrélées grâce à la pénalité de Ridge. Aussi, comme décris ci-dessus, les combinaisons des pénalités de ridge et de Lasso offrent u compromis entre la séléction stricte de variable menée par Lasso et la robustesse face aux colinéarités de Ridge. L'application du modèle ElasticNet sur nos données est donc justifié car cette méthode va permettre de traiter la forte corrélation entre nos variables (par

la pénalité de Ridge), tout en prenant en compte le fait que notre nombre de variable est beaucoup plus important que notre nombre d'observations (par la pénalité de Lasso). A noter : le choix des hyperparamètres par validation croisée sera beaucoup plus long cette fois ci car ElasticNet effectue des itérations non pas pour 1 hyperparamètre, mais pour 2 ($L_1 et L_2$).

Selon nos résultats, la régression ElasticNet avec validation croisée minimise le RMSE de manière efficace (RMSE =). Ce modèle nous à permis d'obtenir le meilleur score sur le data challenge comparé aux scores des autres groupes de cette année, ainsi que de ceux de l'année précédente.

4) Conclusion et ouverture :

Nom du modèle	**Temps d'exécution**	**Nombre de coefficients**	**RMSE (données d'entraînement)**	**RMSE (soumission finale)**
XGBoost	24s		7.0429	7.2084
LinearReg	- de 1s		3.1802	3.1802
RidgeCV	- de 1s	10 000	3.1807	47.76
LassoCV	36s	191	3.2987	4.7915
ElasticNetCV	39s	356	3.1495	4.2682

Conclusion

L'analyse menée a permis d'examiner différents modèles de régression afin de prédire l'age de la population des dnonées en fonction de données de méthylation d'ADN. Le modèle de régression linéaire simple comparé au modèle d'arbres de décisions XGBoost a permis de confirmer l'intuition selon laquelle les relations entre nos variables étaient significativement linéaires. Cette découverte a orienté nos recherches vers des modèles linéaires classiques de régression linéaire pénalisée comem Ridge, Lasso, puis Elasticnet qui est une combinaison des deux. D'une part, la régression de Ridge, bien que performante sur les données d'entrainement, a subi un overfitting marqué, affectant le score de la soumission. La régression de Lasso, de par ses capacités de séléction stricte de variables, a offert nue meilleure généralisation car pénalisant moins les coefficients. Enfin, ElasticNet a pu combiner les avantages des deux pénalités et a pu estimer l'age sur les données de test : ce modèle constitue le meilleur modèle trouvé par notre équipe et ayant fourni le score le plus performant sur ce data challenge. Tout ces résultats n'auront pas pu être atteints aussi rapidement sans l'aide de la recherche d'hyperparamètres optimisée par validation croisée fournie pour les fonctions RidgeCV, LassoCV et ElasticNetCV qui nous ont permis, au dépit du temps de calcul, de trouver les hyperparamètres optimaux afin d'optimiser nos modèles.

Ouverture

1) Influence du choix de la plateforme sur les performances des modèles

Ayant discuté avec d'autres participants du data challenge, nous avons remarqué que, malgré avoir utilisé les m^émes modèles que les notres, d'autres participants ont obtenu des prédictions moins performantes. Ceci peut ètre dù au choix du langage de programmation et des spécificités de ce dernier. En effet, outre les fonctions des packages qui peuvent ètre codés en différents langages (scikit.learn : C, Fortran v.s. glmnet : Fortran), les techniques de grid-search ne sont pas les mêmes pour les deux packages (scikit;learn utilise sa propre méthode configurable là où glmnet utilise une recherche d'hyperparamètres sur une échelle logarithmique). Ces influennces, couplées à l'aléa des simulations not aussi joué un rôle non négligeable dans notre position de premier au classement.

2) Triche

Le nombre de soumissions possibles pour ce data challenge était de 100 : nous aurions pu créer un système d'équations ou un algorithme de prédiction des paramètres optimaux pour minimiser le RMSE en créeant un dataset avec toutes nos soumissions et nos résultats. Cependant, nous participions bien à un data challenge et non pas à un hacking compétition. Ce point reste tout de même important à souligner même si, dans le cadre de ce data challenge, les organisateurs ont laissé ce nombre à 100 en conscience de cause.

3) Source d'inforrmations utilisées pour la rédaction de ce rapport :

Les principales sources utilisées pour générer ce rapport sont :

- Wikipedia
- Documentation officielle scikit-learn
- Utilisation des supports de cours et de TD d'analyse de donneés / GLM / tests statistiques / Python