Entraînez un modèle prédictif linéaire

10 heures Moyenne

Mis à jour le 16/03/2023

TP - Comparez le comportement du lasso et de la régression ridge

Dans ce chapitre, nous allons mettre en pratique les deux algorithmes de régularisation de la régression linéaire étudiés précédemment : la régression ridge et le lasso. Pour cela, nous allons appliquer et comparer les résultats de ces deux algorithmes sur un jeu de données sur le cancer de la prostate.

Chargeons le jeu de données

Le jeu de données est téléchargable **ici**.

C'est un jeu de données assez classiques relativement petit, avec 97 observations et 9 variables. Nous allons ici définir comme target **y** la quantité d'expression de l'antigène qui est associée à la détection de ce cancer (la colonne *lpsa* du dataset). Les autres variables sont des constantes associées, dont le détail est disponible ici

Première étape, charger le jeu de données à l'aide de pandas :

python

```
import pandas as pd
raw_data = pd.read_csv('prostate.data.txt', delimiter='\t')
```

On récupère nos variables explicatives pour la régression, qu'on place dans une matrice X et notre variable expliquée $\(y\)$ à part. On ne récupère pas la dernière colonne du dataset qui est un booléen associé à la présence du cancer, car on ne traite pas les variables discrètes dans ce TP - il nous sera utile pour le TP classification en revanche 😉

python

```
X_train = raw_data.iloc[:60,1:-3]
y_train = raw_data.iloc[:60,-2]
X_test = raw_data.iloc[60:,1:-3]
y_test = raw_data.iloc[60:,-2]
```

Ok on peut maintenant effectuer nos régressions ! 💪

La baseline : une régression classique

La première étape est d'effectuer une régression linéaire classique afin de récupérer une erreur baseline, qu'on souhaite améliorer à l'aide des techniques de régularisation.

python

```
from sklearn import linear_model
import numpy as np

# On crée un modèle de régression linéaire
lr = linear_model.LinearRegression()

# On entraîne ce modèle sur les données d'entrainement
lr.fit(X_train,y_train)

# On récupère l'erreur de norme 2 sur le jeu de données test comme baseline
baseline_error = np.mean((lr.predict(X_test) - y_test) ** 2)

print(baseline_error)
```

On obtient l'erreur quadratique ci-dessous

```
2.8641499657014458
```

Application de la régression ridge

Comme vu dans le chapitre sur la régression ridge, on doit trouver un coefficient de régularisation adapté. Pour rappel, l'objectif est de biaiser un peu la prédiction, afin de diminuer l'erreur standard.

On appelle ce coefficient alpha, on va en tester un certain nombre afin de trouver celui qui est optimal

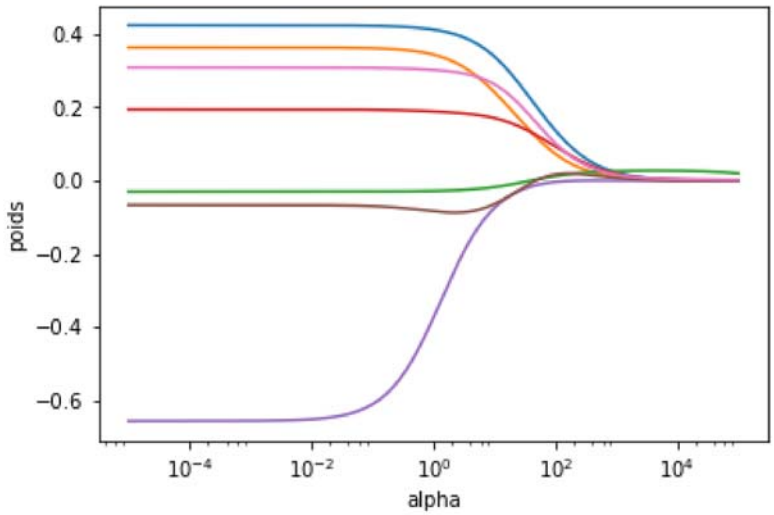
python
```
n_alphas = 200
alphas = np.logspace(-5, 5, n_alphas)
```

On peut maintenant tester toute les régressions ridges avec les différentes valeur de l'hyperparamètre $\(\alpha\)$. On récupère les poids des différents coefficients de la régression associées ainsi que l'erreur quadratique.

python
```
from sklearn.linear_model import Ridge
ridge = linear_model.Ridge()

coefs = []
errors = []
for a in alphas:
    ridge.set_params(alpha=a)
    ridge.fit(X_train, y_train)
    coefs.append(ridge.coef_)
    errors.append([baseline_error, np.mean((ridge.predict(X_test) - y_test) ** 2)])
```

On peut afficher l'évolution de la valeur des différents poids associés aux paramètres

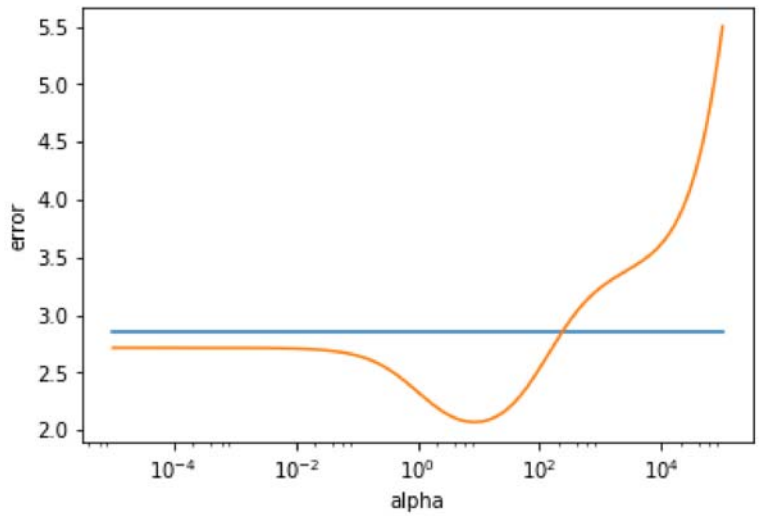
python
```
import matplotlib.pyplot as plt

ax = plt.gca()

ax.plot(alphas, coefs)
ax.set_xscale('log')
plt.xlabel('alpha')
plt.ylabel('weights')
plt.title('Ridge coefficients as a function of the regularization')
plt.axis('tight')
plt.show()
```



Comme on peut le voir (et comme c'était prévu), la valeur de alpha diminue les poids de tous les paramètres de la régression. Etudions maintenant la valeur de l'erreur quadratique

python

```
ax = plt.gca()

ax.plot(alphas, errors)
ax.set_xscale('log')
plt.xlabel('alpha')
plt.ylabel('error')
plt.axis('tight')
plt.show()
```



Comme on peut le voir, la régularisation diminue l'erreur sur le jeu de données test. Pour la régression ridge, le minimum semble se trouver vers alpha=10. On peut récupérer la valeur minimum:

python

```
min(errors)
```

```
2.0722706805123288
```

C'est déjà mieux 🙂

> Pour effectuer une cross validation de la régression ridge, vous pouvez utiliser la fonction `sklearn.linear_model.RidgeCV` qui effectue une recherche automatique des hyperparamètres. J'ai ici effectué une recherche manuelle pour le TP.

Comparons maintenant avec le Lasso !

Application du Lasso

On teste aussi un certain nombre d'hyperparamètres pour appliquer le lasso

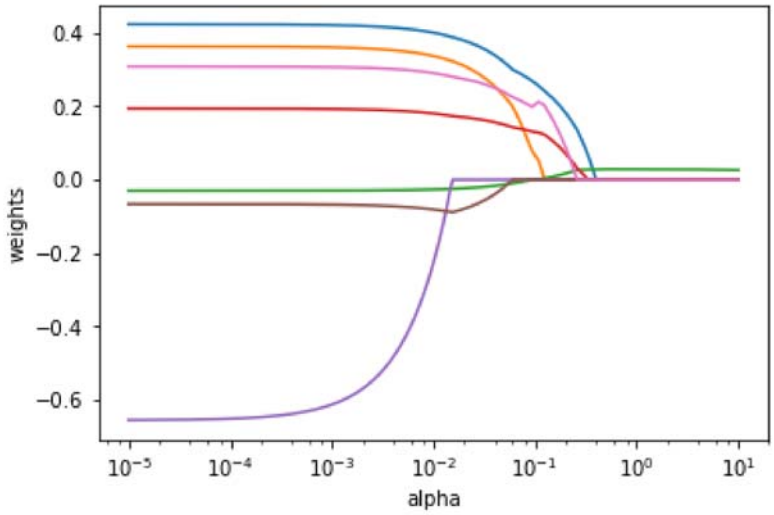
python

```
n_alphas = 300
alphas = np.logspace(-5, 1, n_alphas)
lasso = linear_model.Lasso(fit_intercept=False)

coefs = []
errors = []
for a in alphas:
    lasso.set_params(alpha=a)
    lasso.fit(X_train, y_train)
    coefs.append(lasso.coef_)
    errors.append([baseline_error, np.mean((lasso.predict(X_test) - y_test) ** 2)])
```

On peut maintenant afficher les résultats

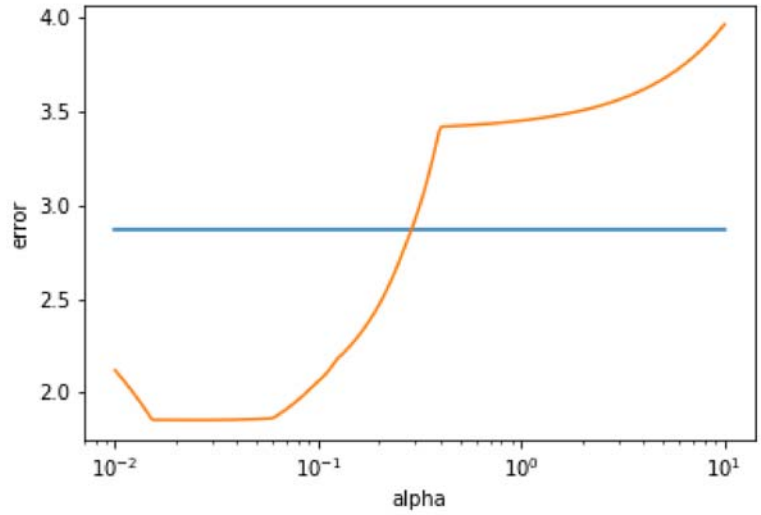
python

```
ax = plt.gca()

ax.plot(alphas, coefs)
ax.set_xscale('log')
plt.xlabel('alpha')
plt.ylabel('weights')
plt.axis('tight')
plt.show()
```



Comme on peut le voir, le lasso permet de supprimer des variables en mettant leur poids à zéro. C'est le cas si deux variables sont corrélées. L'une sera sélectionnée par le Lasso, l'autre supprimée. C'est aussi son avantage par rapport à une régression ridge qui ne fera pas de sélection de variables.

On peut observer maintenant le comportement de l'erreur.



Le minimum est atteint à

```
1.8531561201728328
```

On fait encore mieux qu'avec la régression ridge! En effet comme vu dans les chapitres précédents, le lasso a pour avantage de pouvoir sélectionner un sous-ensemble des variables explicatives afin de permettre une meilleur généralisation.

De même, la fonction `sklearn.linear_model.LassoCV` permet d'effectuer une recherche des hyperparamètres de manière automatisée

Conclusion

Avec ce TP vous avez pu mettre en oeuvre de manière pratique la régression ridge et le lasso. La régularisation fonctionne bien dans les deux cas, et permet de réduire l'erreur totale du modèle à l'aide de l'ajout du biais qui quantifie la complexité. L'un diminue grandement l'influence de certaines variables sur le modèle tandis que l'autre peut directement les supprimer (mettre leur poids à zéro), et donc être parcimonieux.

Réduisez le nombre de variables utilisées par votre modèle

Quiz : Partie 1

Les professeurs

Yannis Chaouche

Newsletter hebdomadaire pour les data scientists - mlacademy.substack.com

Chloé-Agathe Azencott

Chargée de recherche au CBIO de MINES ParisTech Institut Curie. Machine learning ; bioinformatique.