2. Régression : rendement du maïs

Introduction

Le but de cette partie est de prédire les anomalies de rendement du maïs en France, en fonction de plusieurs variables. On commence par explorer le jeu de données. On se rend compte que certaines variables sont factorielles : pour l'année et le département. Les autres variables représentent des mesures de différentes caractéristiques quantitatives à un département et une année donnés. On vérifie également qu'aucune valeur n'est absente dans le jeu de données, et que les variables sont de la bonne classe. On transforme ainsi les variables "year_harvest", "NUMD" et "IRR", considérées par le logiciel comme des variables numériques, en variables qualitatives. On sera parfois amenés à supprimer, par la suite, les variables "année" et "département" avant d'entraîner nos modèles. Ce choix se justifie d'une part par le fait qu'on peut légitimement supposer que l'information apportée par l'année et le département est retranscrite à travers les variables quantitatives, dont les mesures sont effectuées dans un département et à une année précises. D'autre part, leur transformation via un codage disjonctif complet entraînerait une explosion du nombre de paramètres de nos modèles.

Préparation des données et modèles entraînés

On entraîne différents modèles dans cette partie. En fonction du modèle, différents traitements doivent être appliqués à nos données en amont.

- *Régression linéaire et régression de Ridge :* La régression linéaire et la régression de Ridge se basent uniquement sur des variables quantitatives. Ainsi, on supprime l'année, l'indice IRR et le département. On applique nos régressions linéaires sur un jeu de données centré et réduit via la fonction **scale**.

```
scaledData<- scale(data[,-c(1,2,3,4)])
```

- *Decision-tree, Bagging et Random Forest* Ces méthodes ne nécessitent pas de traitement particulier de nos données en amont.
- *GLM :* En plus d'une régression linéaire simple, on essaie aussi d'entraîner une régression linéaire généralisée en prenant en compte les variables qualitatives, qui sont automatiquement transformées en variables muettes par lafonction *glmnet* via un codage dijonctif complet. On centre et réduit les variables quantitatives afin d'équilibrer les poids des différentes variables.
- *GAM et smooth splines :* Les GAM combinés aux Smooth Splines sont coûteux en temps de calcul lorsqu'on veut optimiser l'hyperparamètre de chaque Smooth Spline. Aussi, on décide d'effectuer une PCA sur nos données et de ne garder qu'un certain nombre de composantes principales. Avant cela, on retire de notre jeu de données les variables "année" et "département". On ajoute ensuite les variables muettes issues du codage disjonctif de la variable "IRR".

```
pca <- princomp(scaledData[,-c(1,2,3,4)])
acpdata <- cbind(data[,c(1,2,3,4)],pca$scores)
for(i in 5:ncol(acpdata)) {
  colnames(acpdata)[i] <- paste("Z",i-4,sep="")
}
acpdata <- as.data.frame.matrix(acpdata)
```

- *SVR :* Les SVR sont entraînés sur un jeu de données dont on a retiré l'année et le département, dont les variables quantitatives sont normalisées et auquel on a ajouté le codage disjonctif de la variable "IRR".

Sélection de modèle

On choisit notre meilleur modèle via une 10-fold nested cross-validation. On a donc une cross-validation externe avec, à chaque étape, un jeu de données d'entraînement E et un jeu de données de validation V. À

chaque étape, on effectue les opérations suivantes :

Modèles sans hyperparamètre

On entraîne ces modèles sur E et on calcule leur taux d'erreur *MSE* sur V. C'est le cas de la régression linéaire, mais aussi des randomForest et du Bagging (randomForest en prenant tous les prédicteurs en compte). On aurait pu considérer que le nombre de prédicteurs est un hyperparamètre des randomForest à optimiser, mais on a choisi de garder la valeur usuelle de $\sqrt{p}$.

Modèles avec hyperparamètre

Pour ces modèles, on effectue une 10-fold cross-validation interne en divisant E, à chaque fois, en un jeu d'apprentissage A et un jeu de test T. Cette cross-validation permet d'estimer la meilleure valeur de l'hyperparamètre à fixer. On estime ensuite l'erreur du modèle sur le jeu de donénes de validation V. * Decition-Tree : il faut fixer le coût d'élagage par cross-validation puis couper l'arbre en suivant ce coût. La cross-validation est faite automatiquement via la méthode *rpart*

```
fit.tree <- rpart(y~year_harvest+NUMD, data=data, subset = (folds!=k),  method="anova", control=rpart.c
  i.min<-which.min(fit.tree$cptable[,4])
  tree.cp.opt<-fit.tree$cptable[i.min,1]
  pruned_tree <- prune(fit.tree, cp=tree.cp.opt)
```

- GAM et Smooth Splines : On choisit différents nombres de compoSantes principales, en optimisant à chaque fois le degré de chaque Smooth Spline associé à chaque variable. On retient ensuite le modèle avec la meilleure erreur sur V. Deux paramètres sont donc ici ajustés : le degré des Smooth Splines et le nombre d'axes principaux. La cross-validation interne est représentée ci-après :

```
interndata = acpdata[folds!=k,]
  intern_n <- nrow(interndata)
  intern.Kfolds <- 10
  intern.folds = sample(1:intern.Kfolds, intern_n, replace = TRUE)
  CV.gamSmoothSplines <- rep(0,54)
  for(intern.k in 1:intern.Kfolds) {
    print(intern.k)
    for(i in 1:54) {
      optimalSmoothSplines <- optim(par=rep(i,1), fn = mse.gam.smooth, folds = folds, npcomp=i, control=
      #mse.gam.smooth construit un modèle GAM avec "npcomp" composantes principales
      optimalParam <- optimalSmoothSplines$par
      fit.gamSmoothSplines <- gam(build.formula.smooth(i,optimalParam),data=acpdata[folds!=k,])
      #build.dormula construit une formula avec i composantes principales et optimalParam est le vecteu
      pred.gamSmoothSplines <- predict(fit.gamSmoothSplines,newdata=acpdata[folds==k,],type='response')
      CV.gamSmoothSplines[intern.k] <- CV.gamSmoothSplines[intern.k] + mean((pred.gamSmoothSplines - acp
    }
  }
  optimalnpcomp <- which.min(CV.gamSmoothSplines)
```

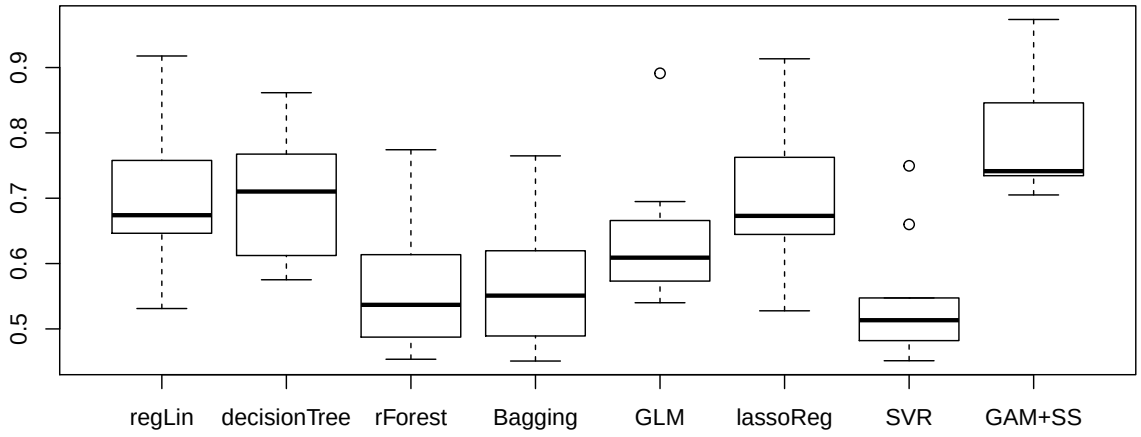
- SVR : Cette fois les apramètres à fixer sont *epsilon* et *c*. n utilise la fonction *tune* pour optimiser ces paramètres sur le jeu de données E. On évalue ensuite l'erreur moyenne sur V.

```
optimal.radial.svr = tune(svm, y~., data=scaledDataDummyIrrWithoutYearAndDep[folds!=k,], ranges=list(eps
fit.radial.svr <- svm(y~., data=scaledDataDummyIrrWithoutYearAndDep[folds!=k,],
epsilon=optimal.radial.svr$best.parameters[,1],cost=optimal.radial.svr$best.parameters[,2])
pred.radial.svr = predict(fit.radial.svr, newdata=scaledDataDummyIrrWithoutYearAndDep[folds==k,-c(1)])
mse.radial.svr[k] <- mean((pred.radial.svr - scaledDataDummyIrrWithoutYearAndDep$y[folds==k])^2)
```

Ajustement et entraînement du modèle choisi

Choix du modèle

La nested cross-validation ci-dessus nous renvoie les estimations des erreurs des différents modèles :



Le modèle qu'on choisit est donc SVR, car il présente les plus petites espérances et variances de MSE.

Ajustement et entraînement du modèle

Il reste maintenant à optimiser les hyperparamètres du SVR. Comme précédemment, on utilise la fonction *tune* pour cela. Elle nous renvoie les meilleurs paramètres $\$epsilon$=0.2, $cost$=2\} Enfin, on entraîne notre modèle avec l'ensemble du jeu de données qui nous a été fournit.

Estimation de l'erreur de prédiction

Pour estimer l'erreur du modèle qu'on a choisi, on réalise une 10-fold cross-validation sur notre jeu de données.

On obtient une erreur de : $MSE = 0.5371496$.