SY19 TP12

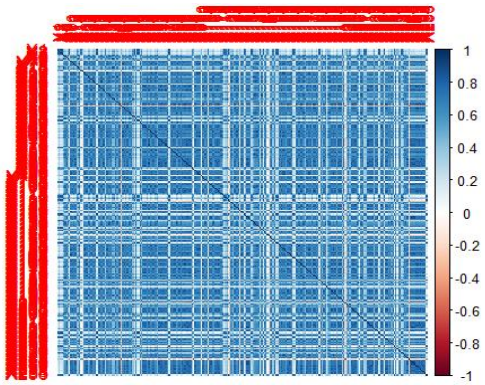
Hou Longhao(contribution 50%), Shen Siyi(contribution 50%)

1. Données Construction

Ce jeu de données comprend 256 prédictuers de log-périodogrammes calculés à partir des trames vocales et 1 variable de réponse ayant 5 classes de phonèmes.

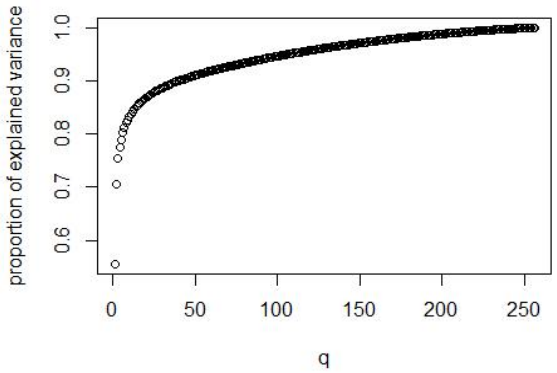
1.1 L'analyse des liens entre les prédicteurs

```
phonemes=read.table("phonemes_train.txt")

cor_matrix = cor(phonemes[,-257])
corrplot(cor_matrix, method = "color")
```



On constate qu'il y a 256 prédicteurs dans cet ensemble de données et qu'il existe une forte corrélation entre les prédicteurs. Par conséquent, tout d'abord, on envisage de faire certains traitements sur les prédicteurs. On utilise PCA(Principal component analysis) pour la sélection.

```
X = phonemes[,1:256]
pca = prcomp(X,center=TRUE,scale=TRUE)
Z = pca$x
lambda = pca$sdev^2
plot(cumsum(lambda)/sum(lambda),xlab="q",ylab="proportion of explained variance")
```



On peut voir sur le graphique que lorsqu'il y a moins de 100 composantes principales, le rapport de variance expliqué est déjà élevé (plus de 90 %), on peut donc simplifier le modèle en conservant les composantes principales appropriées.

```
variance_explained = pca$sdev^2 / sum(pca$sdev^2)
cumulative_variance = cumsum(variance_explained)
components_90_percent = which(cumulative_variance >= 0.9)[1]
print(components_90_percent)
```

```
## [1] 40

final_feature = Z[,1:components_90_percent]
phonemes_pca = as.data.frame(final_feature)
phonemes_pca$y = phonemes$y
```

1.2 Méthode de classer: PCA + KSVM one-svc + KSVM C-svc

Après avoir selectionné les fonctionnalités, car dans ce problème, le modèle formé via l'ensemble d'apprentissage doit être capable d'identifier la catégorie inconnue dans l'ensemble de test et de classer correctement les autres catégories. On entraîne deux modèles de kernel machine à vecteurs de support(KSVM), un pour détecter la nouveauté et d'autre pour classer.

```
#détecter la nouveauté
one_svc_fit = ksvm(y~., data=phonemes_pca,scaled=TRUE, kernel ="rbfdot", kpar = "automatic", type="one-svc",nu=0.15,cross=10)

#Comme on sait que la proportion de classifications inconnues dans l'ensemble de test est de 0.18(500/(500+2259)), on choisit i
ci nu, une limite inférieure à l'erreur de 0.15

#classer
c_svc_fit = ksvm(as.factor(y) ~ ., data = phonemes_pca, scaled = TRUE, type = "C-svc", kernel = "rbfdot",cross = 10,C=100)
```

Ici, on commence par filtrer et marquer les observations anormales sur la base du modèle one_svc_fit, puis on classe les observations anormales dans la classe inconnue et les autres observations sont classées en fonction des résultats du modèle c_svc_fit.

```
prediction_phoneme_1 <- function(dataset){
  library(kernlab)
  data_test_pca = predict(pca,newdata=dataset[,1:256])
  data_test_pca = as.data.frame(data_test_pca[,1:40])
  data_test_pca$y = dataset$y
predictions_oneclass = predict(one_svc_fit,newdata=data_test_pca)
predictions_csvc = predict(c_svc_fit,newdata=data_test_pca)
binary_predictions_oneclass = ifelse(predictions_oneclass == "FALSE","Novel", "Known")
final_predictions = ifelse(binary_predictions_oneclass == "Novel","?",as.character(predictions_csvc))
}

y=prediction_phoneme_1(phonemes)
Tab<-table(y,phonemes$y)
print(Tab)

##
## y     aa  ao dcl  iy  sh
##   ?   101  77   5 192   0
##   aa  302   0   0   0   0
##   ao    0 472   0   0   0
##   dcl   0   0 403   0   0
##   iy    0   0   0 465   0
##   sh    0   0   0   0 483

mean((y==phonemes$y))

## [1] 0.85
```

1.3 Méthode de classer: isolationForest + KSVM C-svc

Dans cette méthode, on repalce KSVM one-svc par isolationForest, aussi une façons pour détecter la nouveauté mais peut-être plus efficace.

```
#détecter la nouveauté
iso_forest = isolationForest$new()
#classer
c_svc_fit = ksvm(as.factor(y) ~ ., data = phonemes, scaled = TRUE, type = "C-svc", kernel = "rbfdot",cross = 10,C=100)

prediction_phoneme_2 <- function(dataset){
  library(solitude)
  library(kernlab)
  iso_forest$fit(dataset[,-257])
  scores = iso_forest$predict(dataset[,-257])
  threshold = quantile(scores$anomaly_score, 0.85) #la proportion d'ensembles de observations normaux: 0.82
  outlier = ifelse(scores$anomaly_score > threshold, TRUE, FALSE)
  svc_pred = predict(c_svc_fit,newdata=dataset)
  final_predictions = ifelse(outlier == TRUE,"?",as.character(svc_pred))
}
```

```
y=prediction_phoneme_2(phonemes)

## INFO  [15:06:50.536] Building Isolation Forest ...
## INFO  [15:06:50.720] done
## INFO  [15:06:50.724] Computing depth of terminal nodes ...
## INFO  [15:06:50.968] done
## INFO  [15:06:51.076] Completed growing isolation forest

Tab<-table(y,phonemes$y)
print(Tab)

##
## y     aa  ao dcl  iy  sh
##   ?   32   5 199  19 119
##   aa 371   0   0   0   0
##   ao   0 544   0   0   0
##   dcl  0   0 209   0   0
##   iy   0   0   0 638   0
##   sh   0   0   0   0 364

mean((y==phonemes$y))

## [1] 0.8504
```

1.4 Réflexions après la réalisation du test

Après avoir testé sur Maggle, on a constaté que les deux modèles n'étaient pas très performants. Tout d'abord, il se peut que le nombre de caractéristiques sélectionnées par PCA ne soit pas suffisant dans l'ensemble de test pour expliquer les principales caractéristiques de l'ensemble de test. Deuxièmement, comme il n'y a pas d'observations d'anomalies dans l'ensemble de données d'entraînement, le modèle entraîné peut ne pas être en mesure de bien discriminer les anomalies.

Par conséquent, on décide d'ajouter des observations inconnues aléatoires directement à l'ensemble de données d'apprentissage, puis d'entraîner le modèle KSVM C-svc et essayer différent type de noyau.

```
random_donne1=matrix(runif(200*256),nrow=200,ncol=256) #200 observations aléatoires uniformément distribués
random_donne2=matrix(rnorm(200*256),nrow=200,ncol=256) #200 observations aléatoires normalement distribués
newdataframe1=as.data.frame(random_donne1)
newdataframe2=as.data.frame(random_donne2)
colnames(newdataframe1)=colnames(phonemes[,-257])
colnames(newdataframe2)=colnames(phonemes[,-257])
newdataframe1$y="?"
newdataframe2$y="?"

combinedDataFrame1=rbind(newdataframe1, phonemes)
combinedDataFrame2=rbind(combinedDataFrame1, newdataframe2)

c_svc_fit = ksvm(as.factor(y) ~ ., data = combinedDataFrame2, scaled = TRUE, type = "C-svc", kernel = "rbfdot",C=100,cross=10)
print(c_svc_fit@cross)


## [1] 0.081034

c_svc_fit_new1 = ksvm(as.factor(y) ~ ., data = combinedDataFrame2, scaled = TRUE, type = "C-svc", kernel = "laplacedot",C=100,c
ross=10)
print(c_svc_fit_new1@cross)


## [1] 0.065862

c_svc_fit_new2 = ksvm(as.factor(y) ~ ., data = combinedDataFrame2, scaled = TRUE, type = "C-svc", kernel = "polydot",C=100,cros
s=10)

print(c_svc_fit_new2)


## [1] 0.11931
```

Le noyau "laplacedot" a le meilleur performance. Ici,on replace le noyau "rbfdot" par "laplacedot", parce que on trouve que le noyau "laplacedot" est plus sensible aux valeurs aberrantes On sélecte ensuite les paramètres pour trouver le modèle le plus performant.

```
CC = c(25,50,75,100,150)
for (i in 1:length(CC)){
  c_svc_fit_new1 = ksvm(as.factor(y) ~ ., data = combinedDataFrame2, scaled = TRUE, type = "C-svc", kernel = "laplacedot",C=CC
[i],cross=10)
  cat(sprintf(" %.3f %.6f\n",CC[i],c_svc_fit_new1@cross))
}
```

```
##  25.000 0.065862
##  50.000 0.067241
##  75.000 0.066552
##  100.000 0.066897
##  150.000 0.068621
```

```
c_svc_fit_new = ksvm(as.factor(y) ~ ., data = combinedDataFrame2, scaled = TRUE, type = "C-svc", kernel = "laplacedot",C=100)
#lorsque C=100 et C=150, l'erreur de validation croisée est la même, on choisit donc C=100 comme coefficient de pénalité


prediction_phoneme_3 <- function(dataset){
  library(kernlab)
  predict(c_svc_fit_new,newdata=dataset)
}

#On simule les observations anormales
random_donne=matrix(runif(500*256,min=-3,max=2),nrow=500,ncol=256)
newdataframe=as.data.frame(random_donne)
colnames(newdataframe)=colnames(phonemes[,-257])
newdataframe$y="?"
Test1=rbind(newdataframe, phonemes)

y1=prediction_phoneme_3(Test1)
Tab=table(y1,Test1$y)
print(Tab)
```

```
##
## y1      ?  aa  ao dcl  iy  sh
##   ?   500   0   0   0   0   0
##   aa    0 403   0   0   0   0
##   ao    0   0 549   0   0   0
##   dcl   0   0   0 408   0   0
##   iy    0   0   0   0 657   0
##   sh    0   0   0   0   0 483
```

```
mean((y1==Test1$y))
```

```
## [1] 1
```

```
random_donne1=matrix(runif(250*256,min=-3,max=2),nrow=250,ncol=256)
random_donne2=matrix(rnorm(250*256,mean=0,sd=3),nrow=250,ncol=256)
newdataframe1=as.data.frame(random_donne1)
newdataframe2=as.data.frame(random_donne2)
colnames(newdataframe1)=colnames(phonemes[,-257])
colnames(newdataframe2)=colnames(phonemes[,-257])
newdataframe1$y="?"
newdataframe2$y="?"
combinedDataFrame1=rbind(newdataframe1, phonemes)
Test2=rbind(combinedDataFrame1, newdataframe2)

y2=prediction_phoneme_3(Test2)
Tab=table(y2,Test2$y)
print(Tab)
```

```
##
## y2      ?  aa  ao dcl  iy  sh
##   ?   500   0   0   0   0   0
##   aa    0 403   0   0   0   0
##   ao    0   0 549   0   0   0
##   dcl   0   0   0 408   0   0
##   iy    0   0   0   0 657   0
##   sh    0   0   0   0   0 483
```

```
mean((y2==Test2$y))
```

```
## [1] 1
```

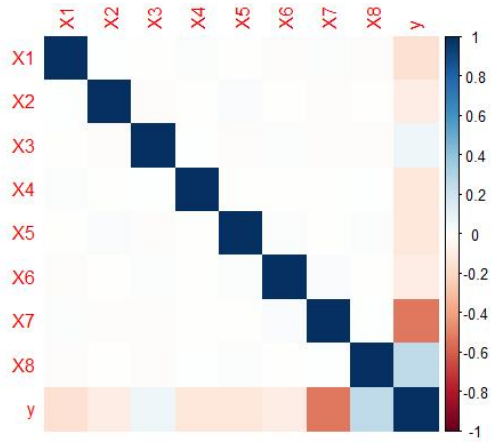
On observe que ce modèle peut identifier efficacement les observations anormales et les classer correctement.

2. EX2 Données Robotics

Ce jeu de données contient des données liées à la cinématique d'un bras de robot.

2.1 L'analyse des liens entre les prédicteurs

```
robot=read.table("robotics_train.txt")
cor_matrix = cor(robot)
corrplot(cor_matrix, method = "color")
```



On peut observer qu'il n'y a aucune relation entre les 8 prédicteurs.

2.2 Régression linéaire avec cross-validation et BIC,AIC

```
fit_reg=lm(y~.,data=robot)
mse = mean((fitted(fit_reg)-robot$y)^2)
print(mse)

## [1] 0.1885955

k=10
n=nrow(robot)
fold=sample(k,n,replace=TRUE)
rms=rep(0,k)
rms.aic=rep(0,k)
rms.bic=rep(0,k)
for(m in 1:k){
  fit=lm(y~.,data=robot,subset= fold != m)
  fit.aic=stepAIC(fit,y~.,direction="both",trace=0)
  fit.bic=stepAIC(fit,y~.,direction="both",trace=0,k=log(sum(fold != m)))
  pred=predict(fit,newdata = robot[fold==m,])
  pred.aic=predict(fit.aic,newdata = robot[fold==m,])
  pred.bic=predict(fit.bic,newdata = robot[fold==m,])
  rms[m]=sqrt(mean((robot$y[fold==m]-pred)^2))
  rms.aic[m]=sqrt(mean((robot$y[fold==m]-pred.aic)^2))
  rms.bic[m]=sqrt(mean((robot$y[fold==m]-pred.bic)^2))
}

print(c(mean(rms),sd(rms/sqrt(k))))

## [1] 0.43489943 0.00335525

print(c(mean(rms.aic),sd(rms.aic/sqrt(k))))

## [1] 0.43489943 0.00335525

print(c(mean(rms.bic),sd(rms.bic/sqrt(k))))

## [1] 0.43489943 0.00335525
```

On observe que l'utilisation de l'AIC et du BIC n'a pas d'effet très significatif sur ce problème.

2.3 Lasso

```
xtrain=as.matrix(robot[,-9])
ytrain=robot[,9]
cv.out=cv.glmnet(xtrain,ytrain,alpha=1)
fit_lasso=glmnet(xtrain,ytrain,lambda=cv.out$lambda.min,alpha=1)
lasso.pred=predict(fit_lasso,lambda=cv.out$lambda.min,newx=xtrain)
mse_lasso=mean((lasso.pred-ytrain)^2)
print(mse_lasso)

## [1] 0.1886013
```

Le lasso et la régression linéaire ont des effets similaires.

2.4 KSVM nu-svr

```
svmfit_cv=ksvm(y~.,data=robot,scaled=TRUE,type="nu-svr",kernel="rbfdot",C=100,nu=0.1,cross=10)
print(svmfit_cv)

## Support Vector Machine object of class "ksvm"
## SV type: nu-svr  (regression)
##  parameter : epsilon = 0.1  nu = 0.1
## Gaussian Radial Basis kernel function.
##  Hyperparameter : sigma =  0.0856642772283093
## Number of Support Vectors : 1286
## Objective Function Value : -21148.28
## Training error : 0.045594
## Cross validation error : 0.028128
```

On trouve que le modèle KSVM est significativement plus performant. On sélecte ensuite les paramètres pour trouver le modèle le plus performant.

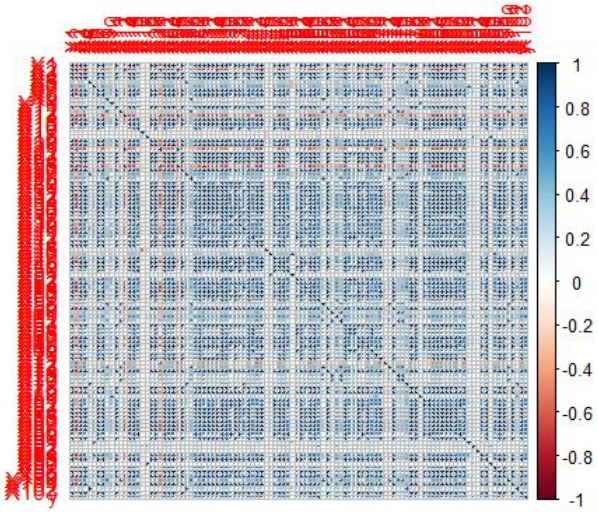
```
CC = c(25,50,75,100)
NU = c(0.1,0.15,0.2,0.25,0.3)

for (i in 1:length(CC)){
  for (j in 1:length(NU)){
    svmfit_cv=ksvm(y~.,data=robot,scaled=TRUE,type="nu-svr",kernel="rbfdot",C=CC[i],nu=NU[j],kpar=list(sigma=0.1),cross=10)
    cat(sprintf(" %.3f %.3f %.6f\n",CC[i],NU[j],svmfit_cv@cross))
  }
}

##  25.000 0.100 0.028373
##  25.000 0.150 0.026881
##  25.000 0.200 0.025771
##  25.000 0.250 0.025182
##  25.000 0.300 0.025114
##  50.000 0.100 0.027081
##  50.000 0.150 0.026018
##  50.000 0.200 0.025525
##  50.000 0.250 0.025170
##  50.000 0.300 0.024688
##  75.000 0.100 0.027648
##  75.000 0.150 0.026126
##  75.000 0.200 0.025176
##  75.000 0.250 0.024729
##  75.000 0.300 0.024694
##  100.000 0.100 0.027311
##  100.000 0.150 0.026256
##  100.000 0.200 0.025405
##  100.000 0.250 0.025264
##  100.000 0.300 0.025910
```

On peut observer que avec la même valeur de nu, à partir de C=50, à mesure que C augmente, l'erreur de validation croisée devient plus grande. On choisit donc C=50 comme coefficient de pénalité. Et dans les tests, on commencer à mettre nu=0.1 et l'augmenter progressivement la valeur nu.

3. Données Construction

Ce jeu de données comprend le coût de construction, le prix de vente, les variables du projet et les variables économiques correspondant aux appartements résidentiels à Téhéran, en Iran.

```
construction=read.table("construction_train.txt")
construction.cor=cor(construction)
corrplot(construction.cor)
```



3.1. Analyse exploratoire

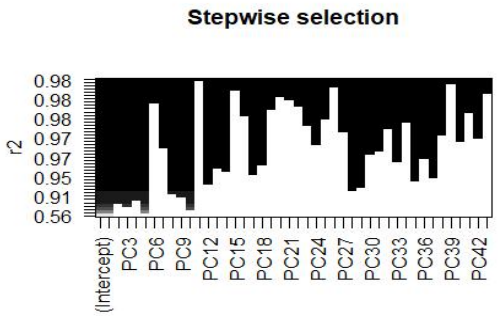
Le dataset contient 103 prédicteurs (X1, X2, ⋯, X103), et une colonne y. Le fichier contient 250 observations. Les prédicteurs sont fortement corrélés donc il est nécessaire de réaliser une ACP. On sélectionne les composantes principales qui conservent 99,9 % de la variance, et il ne reste que 43.

```
pca_con=prcomp(construction[,1:103],scale=TRUE)
cumulative_variance <- cumsum(pca_con$sdev^2) / sum(pca_con$sdev^2)
num_components <- which(cumulative_variance >= 0.999)[1]
nn=num_components
data_pca <- as.data.frame(pca_con$x[, 1:num_components])
data_pca$y=construction$y
```

3.2. Recherche du modèle

Pour trouver un meilleur modèle, on fait des régressions sur les sous-ensembles de prédicteur sélectionné par la méthode pas à pas exhaustive.

```
reg.fit <-regsubsets(y ~ .,data=data_pca,method="exhaustive",nvmax=nn)
reg.subsets <- summary(reg.fit)
```



On calcule la MSE de la régression linéaire, la régression ridge et la régression lasso. On fait la cross-validation en 5 parties, calcule l'erreur de test en prenant chacun de ces sous-ensembles comme ensemble de test et les autres comme ensemble d'apprentissage, et on calcule la MSE. On répète 10 fois et calcule la moyenne.

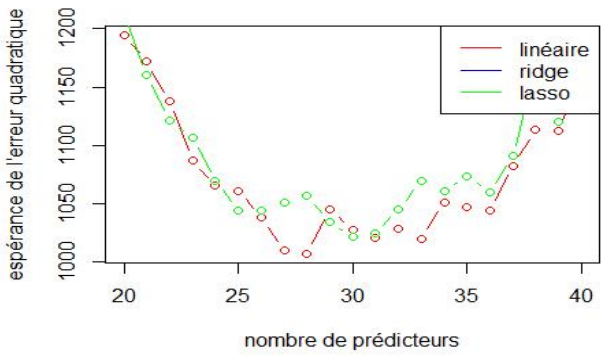
```
n.reg = dim(data_pca)[1]
p.reg = dim(data_pca)[2]
reg.models.list <- c("lm", "ridge", "lasso") # Liste des modeles
reg.subset.error <- c()
reg.models.error <- matrix(0,nrow = nn-2,ncol = 3)
colnames(reg.models.error) <- reg.models.list
rownames(reg.models.error) <- c(2:(nn-1))
reg.error.model <- function(model, reg.n_folds, reg.trials, which.subsets){
```

```
  for (subset in which.subsets){
    reg.subset.error[subset] <- 0
    for(i in c(1:reg.trials)) {
      reg.folds <- sample(rep(1:reg.n_folds, length.out=n.reg))
      for(k in 1:reg.n_folds){
        reg.train <- data_pca[reg.folds!=k,]
        reg.train.X <- reg.train[,-p.reg]
        reg.train.y <- reg.train$y
        reg.test <- data_pca[reg.folds==k,]
        reg.test.X <- reg.test[,-p.reg]
        reg.test.y <- reg.test$y
        reg.subset <- reg.train.X[,reg.subsets$which[subset,2:(nn+1)]]
        reg.subset.test <- reg.test.X[,reg.subsets$which[subset,2:(nn+1)]]
        reg.model <- NULL
        if(model == "lm"){
          reg.model <- lm(reg.train.y ~., data = reg.subset)
        }
        else{
          if(model == "ridge"){
            cv.out<-cv.glmnet(as.matrix(reg.subset),reg.train.y,alpha=0)
            reg.model<-glmnet(as.matrix(reg.subset),reg.train.y,lambda=cv.out$lambda.min,alpha=0)
          }
          else if(model == "lasso"){
            cv.out<-cv.glmnet(as.matrix(reg.subset),reg.train.y,alpha=1)
            reg.model<-glmnet(as.matrix(reg.subset),reg.train.y,lambda=cv.out$lambda.min,alpha=1)
          }
          else{
            reg.model <- NULL
          }
        }

        if(!is.null(reg.model)){
          if(model == "lm"){
            reg.predict <- predict(reg.model, newdata = reg.subset.test)
            reg.subset.error[subset] <- reg.subset.error[subset] + mean((reg.test.y - reg.predict)^2)
          }
          else{
            reg.predict <- predict(reg.model,s=cv.out$lambda.min,newx=as.matrix(reg.subset.test))
            reg.subset.error[subset] <- reg.subset.error[subset] + mean((reg.test.y - reg.predict)^2)
          }
        }
      }
    }
    reg.subset.error[subset] <- reg.subset.error[subset]/(reg.n_folds*reg.trials)
  }
  reg.subset.error
}
for (model in reg.models.list){
  reg.models.error[,model] <- as.vector(reg.error.model(model, 5,10, c(2:(nn-1)))[2:(nn-1)])
}
```

On observe que la MSE en fonction du nombre de prédicteurs diminue jusqu'à atteindre un minimum pour un nombre de prédicteurs compris dans l'intervalle [40,60].

On répète donc en se limitant aux sous-ensembles de 40 à 60 prédicteurs avec une répétition 30 fois.

```
reg.subset.error <- c()
reg.models.error.zoom <- matrix(0,nrow = 21,ncol = 3)
colnames(reg.models.error.zoom) <- reg.models.list
rownames(reg.models.error.zoom) <- c(20:40)
for (model in reg.models.list){
  reg.models.error.zoom[,model] <- as.vector(reg.error.model(model, 5,30, c(20:40))[20:40])
}
```



```
a=which.min(reg.models.error.zoom[,1])
b=which.min(reg.models.error.zoom[,2])
c=which.min(reg.models.error.zoom[,3])

## 28 27 30
##  9  8 11
```

La MSE la plus faible est obtenue pour 28 prédicteurs avec la régression linéaire, 27 prédicteurs avec la régression ridge et 30 prédicteurs avec la régression lasso. On teste encore sur ces trois modèles avec une répétition 50 fois.

```
##         espérance
## lm28     998.9082
## ridge27 1203.8883
## lasso30 1017.5852
```

On remarque que la plus faible MSE est encore la régression linéaire sur sous-ensembles des 28 prédicteurs. On le choisi pour le tester sur maggle.

```
data_fit=data_pca[,reg.subsets$which[28, 2:(nn+1)]]
construction.fit1=lm(y~.,data=data_fit)
```

3.3. D'autres méthodes

Avec l'idée similaire, on peut faire la régression svm aussi. On supprime les prédicteurs inutiles.

```
cv_fit <- cv.glmnet(x_construction, y_construction, alpha = 1)
fit <- glmnet(x_construction, y_construction, alpha=1, lambda=cv_fit$lambda.min)
coefficients <- coef(fit, s = cv_fit$lambda.min)
non_zero_coef_indices <- which(coefficients != 0)-1
variable_names <- colnames(x_construction)[non_zero_coef_indices]
new_data <- construction[, c(variable_names, 'y')]
```

Ensuite on trouve que le meilleur C et epsilon sont 1 et 0.001.

```
Cs <- c(0.01, 0.1, 1, 10, 100, 1000)
epsilons <- c(0.001, 0.01, 0.1, 1)
bestMSE <- Inf
bestC <- NA
bestEpsilon <- NA
for (C in Cs) {
    for (epsilon in epsilons) {
        MSE_SVM <- rep(0, K)
        for (k in 1:K) {
            train_data <- new_data[fold != k, ]
            test_data <- new_data[fold == k, ]
            svmfit <- ksvm(y ~ ., data = train_data, scaled = TRUE, type = "nu-svr", kernel = "vanilladot", C = C, epsilon = epsi
lon)
            pred <- predict(svmfit, newdata = test_data)
            MSE_SVM[k] <- mean((pred - test_data$y)^2)
        }
        avgMSE <- mean(MSE_SVM)
        if (avgMSE < bestMSE) {
            bestMSE <- avgMSE
            bestC <- C
            bestEpsilon <- epsilon
        }
    }
}
print("C:", bestC, "epsilon:", bestEpsilon)

## C: 1 epsilon: 0.001
```

On applique de nouvelles données et paramètres et obtenu un bas mse.

```
MSE <- rep(0, 10)
for (k in 1:10) {
    train_data <- new_data[fold != k, ]
    test_data <- new_data[fold == k, ]
    fit <- ksvm(y ~ ., data = train_data, scaled = TRUE, type = "nu-svr", kernel = "vanilladot", C = 1, epsilon = 0.001)
    pred <- predict(fit, newdata = test_data)
    MSE [k] <- mean((pred - test_data$y)^2)
}
```

```
print(mean(MSE))

## [1] 1037.211
```

On obtient donc notre deuxième modèle :

```
construction.fit2 = ksvm(y ~ ., data = new_data, scaled = TRUE, type = "nu-svr", kernel = "vanilladot", C = 1, epsilon = 0.001)
```

De plus, on a encore essayé d'autre noyaux et les méthodes de KNN, GAM, mais leurs performances ne sont pas agréables. On a donc finalisé ces deux modèles.

4. Données Images

Il s'agit d'images naturelles en couleur au format JPEG représentant des voitures, des fruits et des chiens. Les images sont de tailles variables. La tâche consiste à prédire le contenu de nouvelles images appartenant à l'un de ces trois types.

Voici des exemples des trois catégories d'images :

4.1. Définition du modèle

On utilise le modèle réseau neuronal CNN avec le package keras pour développer le classifieur.

Comme le que le prétraitement de l'image compressera l'image très petite, la taille du modèle ne sera pas très grande. On a directement essayé le modèle classique vgg19 :

```
img_width <- 32
img_height <- 32
model.image <- keras_model_sequential() %>%
  #layer 1-2
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu", padding = "same",
                input_shape = c(img_width, img_height, 3)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu", padding = "same") %>%
  layer_max_pooling_2d(pool_size = c(2, 2), strides = c(2, 2)) %>%
  #layer 3-4
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu", padding = "same") %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu", padding = "same") %>%
  layer_max_pooling_2d(pool_size = c(2, 2), strides = c(2, 2)) %>%
  #layer 5-8
  layer_conv_2d(filters = 256, kernel_size = c(3, 3), activation = "relu", padding = "same") %>%
  layer_conv_2d(filters = 256, kernel_size = c(3, 3), activation = "relu", padding = "same") %>%
  layer_conv_2d(filters = 256, kernel_size = c(3, 3), activation = "relu", padding = "same") %>%
  layer_conv_2d(filters = 256, kernel_size = c(3, 3), activation = "relu", padding = "same") %>%
  layer_max_pooling_2d(pool_size = c(2, 2), strides = c(2, 2)) %>%
  #layer 9-12
  layer_conv_2d(filters = 512, kernel_size = c(3, 3), activation = "relu", padding = "same") %>%
  layer_conv_2d(filters = 512, kernel_size = c(3, 3), activation = "relu", padding = "same") %>%
  layer_conv_2d(filters = 512, kernel_size = c(3, 3), activation = "relu", padding = "same") %>%
  layer_conv_2d(filters = 512, kernel_size = c(3, 3), activation = "relu", padding = "same") %>%
  layer_max_pooling_2d(pool_size = c(2, 2), strides = c(2, 2)) %>%
  #layer 13-16
  layer_conv_2d(filters = 512, kernel_size = c(3, 3), activation = "relu", padding = "same") %>%
  layer_conv_2d(filters = 512, kernel_size = c(3, 3), activation = "relu", padding = "same") %>%
  layer_conv_2d(filters = 512, kernel_size = c(3, 3), activation = "relu", padding = "same") %>%
  layer_conv_2d(filters = 512, kernel_size = c(3, 3), activation = "relu", padding = "same") %>%
  layer_max_pooling_2d(pool_size = c(2, 2), strides = c(2, 2)) %>%
  #flatten
  layer_flatten() %>%
  #layer 17
  layer_dense(units = 4096, activation = "relu") %>%
  layer_dropout(rate = 0.5) %>%
  #layer 18
  layer_dense(units = 4096, activation = "relu") %>%
  layer_dropout(rate = 0.5) %>%
```

```
  # layer 19, 3 classes
  layer_dense(units = 3, activation = "softmax")

model.image %>% compile(
  loss = "sparse_categorical_crossentropy",
  optimizer = optimizer_rmsprop(learning_rate = 1e-4,weight_decay = 1e-6),
  metrics = "accuracy"
)

summary(model.image)

## Model: "sequential"
## _________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## =========================================================================
##  conv2d_15 (Conv2D)                 (None, 32, 32, 64)              1792
##  conv2d_14 (Conv2D)                 (None, 32, 32, 64)              36928
##  max_pooling2d_4 (MaxPooling2D)      (None, 16, 16, 64)              0
##  conv2d_13 (Conv2D)                 (None, 16, 16, 128)             73856
##  conv2d_12 (Conv2D)                 (None, 16, 16, 128)             147584
##  max_pooling2d_3 (MaxPooling2D)      (None, 8, 8, 128)               0
##  conv2d_11 (Conv2D)                 (None, 8, 8, 256)               295168
##  conv2d_10 (Conv2D)                 (None, 8, 8, 256)               590080
##  conv2d_9 (Conv2D)                  (None, 8, 8, 256)               590080
##  conv2d_8 (Conv2D)                  (None, 8, 8, 256)               590080
##  max_pooling2d_2 (MaxPooling2D)      (None, 4, 4, 256)               0
##  conv2d_7 (Conv2D)                  (None, 4, 4, 512)               1180160
##  conv2d_6 (Conv2D)                  (None, 4, 4, 512)               2359808
##  conv2d_5 (Conv2D)                  (None, 4, 4, 512)               2359808
##  conv2d_4 (Conv2D)                  (None, 4, 4, 512)               2359808
##  max_pooling2d_1 (MaxPooling2D)      (None, 2, 2, 512)               0
##  conv2d_3 (Conv2D)                  (None, 2, 2, 512)               2359808
##  conv2d_2 (Conv2D)                  (None, 2, 2, 512)               2359808
##  conv2d_1 (Conv2D)                  (None, 2, 2, 512)               2359808
##  conv2d (Conv2D)                    (None, 2, 2, 512)               2359808
##  max_pooling2d (MaxPooling2D)        (None, 1, 1, 512)               0
##  flatten (Flatten)                  (None, 512)                     0
##  dense_2 (Dense)                    (None, 4096)                    2101248
##  dropout_1 (Dropout)                (None, 4096)                    0
##  dense_1 (Dense)                    (None, 4096)                    16781312
##  dropout (Dropout)                  (None, 4096)                    0
##  dense (Dense)                      (None, 3)                       12291
## =========================================================================
## Total params: 38919235 (148.47 MB)
## Trainable params: 38919235 (148.47 MB)
## Non-trainable params: 0 (0.00 Byte)
## _________________________________________________________________________
```

4.2. Prétraitement et entraînement

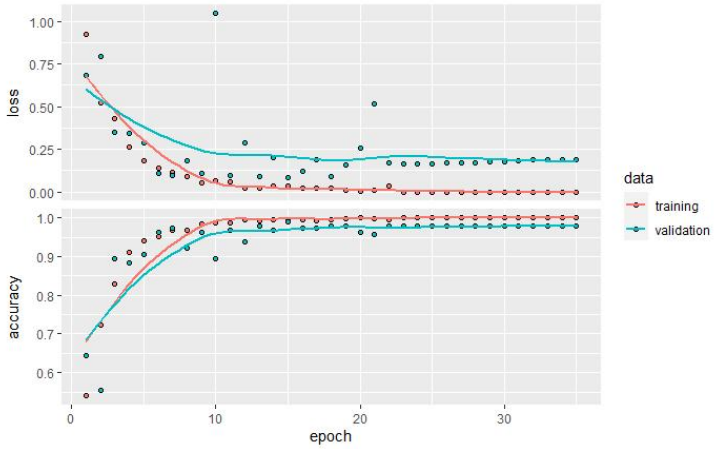
Les images sont simples, donc il n'y a pas besoin de faire une augmentation des données pour varier des images. Après plusieurs tentatives, il nous semble que 35 epochs sont fortement suffisants. On a sélectionné 60 images pour chaque catégorie comme images de validation.

```
train_data <- image_dataset_from_directory(
  train_path,
  image_size = c(img_width, img_height),
  batch_size = 20,
)

## Found 1587 files belonging to 3 classes.

validation_data <- image_dataset_from_directory(
  validation_path,
  image_size = c(img_width, img_height),
  batch_size = 20,
)

## Found 180 files belonging to 3 classes.

history <- model.image %>% fit(
  train_data,
  epochs = 35,
  validation_data = validation_data,
```

```
  shuffle = TRUE,
  verbose = 2
)
```

4.3. Résultat

```
Epoch 1/35
80/80 - 15s - loss: 0.9252 - accuracy: 0.5394 - val_loss: 0.6826 - val_accuracy: 0.6444 - 15s/epoch - 193ms/step
Epoch 6/35
80/80 - 13s - loss: 0.1386 - accuracy: 0.9515 - val_loss: 0.1069 - val_accuracy: 0.9611 - 13s/epoch - 161ms/step
Epoch 11/35
80/80 - 13s - loss: 0.0599 - accuracy: 0.9855 - val_loss: 0.0948 - val_accuracy: 0.9667 - 13s/epoch - 164ms/step
Epoch 16/35
80/80 - 13s - loss: 0.0201 - accuracy: 0.9962 - val_loss: 0.1215 - val_accuracy: 0.9722 - 13s/epoch - 159ms/step
Epoch 21/35
80/80 - 13s - loss: 0.0124 - accuracy: 0.9987 - val_loss: 0.5173 - val_accuracy: 0.9556 - 13s/epoch - 160ms/step
Epoch 26/35
80/80 - 13s - loss: 3.8332e-06 - accuracy: 1.0000 - val_loss: 0.1677 - val_accuracy: 0.9778 - 13s/epoch - 160ms/step
Epoch 31/35
80/80 - 13s - loss: 1.5835e-06 - accuracy: 1.0000 - val_loss: 0.1817 - val_accuracy: 0.9778 - 13s/epoch - 159ms/step
Epoch 35/35
80/80 - 13s - loss: 1.0309e-06 - accuracy: 1.0000 - val_loss: 0.1882 - val_accuracy: 0.9778 - 13s/epoch - 161ms/step
```



D'après les enregistrements, on peut voir que la précision de la classification est très élevée. Pour éviter le surapprentissage, on peut choisir de réduire epochs à 20 ou d'utiliser EarlyStopping.

De plus, comme une tâche simple, le modèle est vraiment gros (148.47Mo). En fait, le modèle avec moins de couches peut atteindre un bon résultat aussi.

On a également essayé le modèle ResNet50(Residual neural network with 50 layers), qui est connu pour sa profondeur et son architecture innovante utilisant des blocs résiduels, facilitant l'entraînement des réseaux profonds en évitant les problèmes de disparition de gradient. Le modèle est plus petit, mais les résultats sont également très bons.

```
## Model: "resnet50"
## ________________________________________________________________________________
## Total params: 23593859 (90.00 MB)
## Trainable params: 23540739 (89.80 MB)
## Non-trainable params: 53120 (207.50 KB)
## ________________________________________________________________________________
```