TP7 Khalis - Mekkoudi - Collas

```
source(file="cv_classif.R")
```

Astronomy

Chargement des données

```
astronomy <- read.table('data/astronomy_train.csv', sep = ",", header = TRUE)
```

Nous commençons par charger les données du jeu de données 'astronomy.csv'. Ce problème est un problème de classification comportant 3 classes. Ce jeu de données comporte 5000 individus décrits par 17 variables. Toutes les variables sont quantitatives. La proportion du nombre d'individus pour chaque classe est la suivante:

	Proportions
GALAXY	0.5038
QSO	0.0836
STAR	0.4126

Séparation des données en entrainement/test

Nous séparons les données en un ensemble d'entrainement sur lequel nous allons tester tout nos modèles et un ensemble de test qui nous permettra d'évaluer l'erreur pour le modèle que nous aurons retenu. Afin de conserver la même séparation des données à chaque execution du notebook nous fixons le générateur de nombres aléatoires (seed).

```
set.seed(42)
train_size <- nrow(astronomy) * (2/3)
train <- sample(1:nrow(astronomy), size = train_size)
astronomy_train <- astronomy[train,]
astronomy_test <- astronomy[-train,]
ncol(astronomy_train)
```

```
## [1] 18
```

```
nrow(astronomy_train)
```

```
## [1] 3333
```

```
nrow(astronomy_test)
```

```
## [1] 1667
```

Transformations des données d'entrainement

Nous appliquons plusieurs transformations à l'ensemble d'entrainement:

Les variables 'objid' et 'rerun' n'ont qu'une seule valeur. Nous les retirons puisqu'elles ne nous servirons pas à discriminer les 3 classes.

Nous centrons et réduisons les variables. En effet certaines variables comme 'specobjid' sont en 10^18 alors que d'autres variables sont en 10^-4 comme 'redshift'.

Nous affichons 3 individus à titre d'exemple:

ra	dec	u	g	r	i	z	run	camcol	field	specobjid	redshift	plate	mjd	fiberid	y
-0.3	-0.5	0.3	-0.3	-0.6	-0.7	-0.7	-0.8	1.4	0.0	-0.5	-0.2	-0.5	-0.6	-0.7	GALAXY
1.6	-0.6	-2.0	-1.7	-1.3	-1.0	-0.8	-0.9	0.8	1.9	-0.6	-0.4	-0.6	-0.8	0.9	STAR
0.5	-0.6	1.1	1.0	0.8	0.8	0.7	1.3	0.2	-1.7	3.4	-0.4	3.4	2.5	2.6	STAR

Méthode d'évaluation

Nous évaluons chaque modèle sur l'ensemble d'entrainement à l'aide d'une validation croisée (avec 10 parties). Nous calculons l'erreur moyenne et l'écart type. Nous choisissons le modèle ayant la plus faible erreur de classification pour notre modèle final. Enfin, nous estimons l'erreur de notre modèle final sur l'ensemble de test.

Validation croisée

Nous avons implémenté une fonction qui calcule l'erreur de classification et l'écart type par validation croisée pour l'ensemble des modèles que nous utiliserons par la suite. Cette fonction s'appelle 'CV_eval'.

NB: Nous utilisons la constante 'RECHERCHE_HYPERPARAMETRES' afin de ne pas rechercher les hyperparamètres à chaque execution de ce notebook.

```
RECHERCHE_HYPERPARAMETRES <- TRUE
```

Analyses discriminantes

```
set.seed(123)
#validation croisée sur l'ensemble d'entrainement
res <- data.frame("Erreur"=rep(0,3), "Ecart type"=rep(0,3))
res[1,] <- CV_eval('adl', astronomy_train)
res[2,] <- CV_eval('adq', astronomy_train)
res[3,] <- CV_eval('bayesien_naif', astronomy_train)
```

Nous obtenons les erreurs suivantes:

	Erreur	Ecart type
ADL	0.0906091	0.0124512
ADQ	0.0276028	0.0091547
Bayesien naif	0.0732073	0.0192217

Nous ne testons pas la régression la régression logistique puisque la'analyse discriminante linéaire donne de mauvaises performances.

L'analyse discriminante quadratique est l'algorithme le plus performant des trois modèles testés précédemment. Les modèles plus complexes fonctionnent donc bien sur ce jeu de données. Nous testons donc SVM avec différents noyaux.

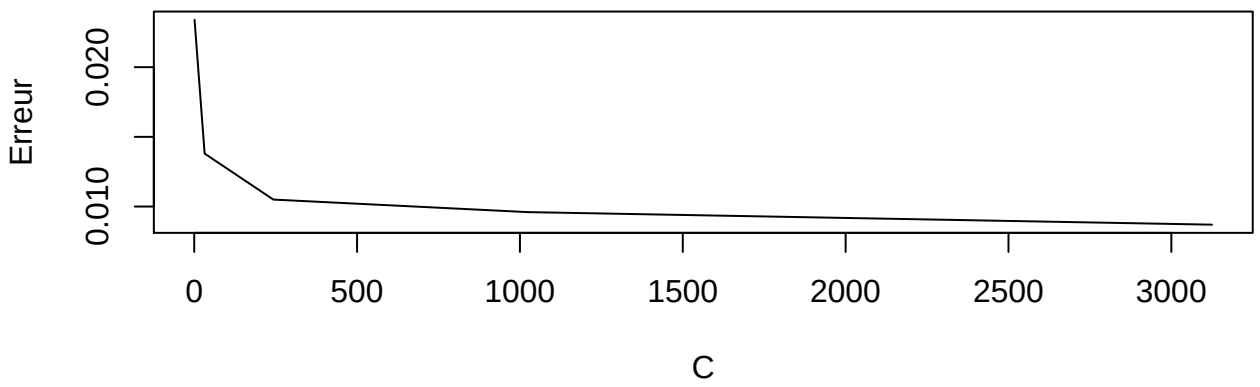
SVM

Optimisation C

Les SVM ont l'hyperparamètre C: si C est faible alors le SVM aura une large marge quitte à mal classer certains points (on peut dans ce cas se retrouver dans un cas de sous entrainement). De manière opposée, si C est grand alors le SVM classifira bien beaucoup de points mais aura une faible marge et donc risque un surentrainement. Nous cherchons le meilleur compromis.

Nous cherchons l'hyperparamètre C minimisant l'erreur de classification pour un noyau linéaire. Nous testons différents C: 1, 32, 243, 1024 et 3125. Pour chaque valeur nous calculons l'erreur sur le jeu d'entrainement par validation croisée.

```
if (RECHERCHE_HYPERPARAMETRES){
  set.seed(123)
  C_list <- (1:5)^5
  erreurs_C <- rep(0, length(C_list))
  for (i in 1:length(C_list)){
    hyperparametres <- list(kernel="vanilla", C=C_list[i], kpar=list())
    erreurs_C[i] <- CV_eval('svc', astronomy_train, hyperparametres)[1]
  }
  plot(C_list, erreurs_C, type="l",xlab="C", ylab="Erreur")
  C <- C_list[which.min(erreurs_C)]
}else{
  C <- 3125
}
```



La valeur de C minimisant l'erreur est 3125.

SVM à noyaux

Les différents noyaux vont nous permettre de trouver des fontières de décision non linéaires.

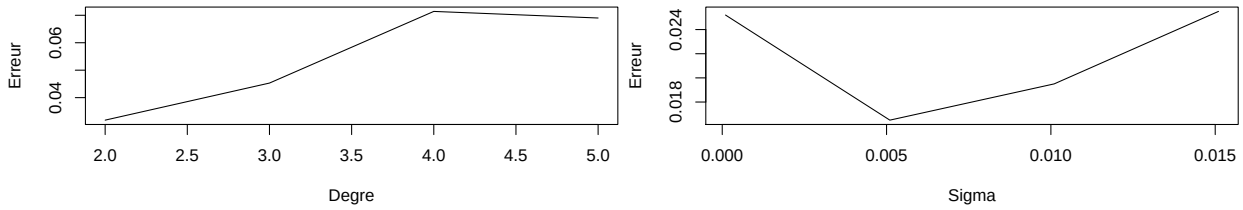
Nous testons trois noyaux pour les SVM:

1. noyau linéaire ('vanilladot') qui n'a pas d'hyperparamètres
2. noyau polynomial pour lequel il faut régler le degré du noyau. Nous testons cet hyperparamètre pour des valeurs de 2 à 5. Nous retenons le degré donnant la plus faible erreur par validation croisée.
3. noyau gaussien pour lequel il faut régler le paramètre sigma. Nous testons cet hyperparamètre pour des valeurs de 0.0001 à 0.02 par pas de 0.005. Nous retenons le sigma donnant la plus faible erreur par validation croisée.

L'idéal serait d'optimiser C pour différents noyaux. Cependant chaque noyau possède d'autres hyperparamètres et essayer toutes les combinaisons se révèle trop couteux en calculs. Par conséquent nous conservons la valeur trouvée avec le noyau linéaire.

Nous observons que le noyau polynomial n'améliore pas les performances de notre classifieur: plus le degré du noyau est élevé plus les performances se dégradent.

De même le noyau gaussien n'améliore pas les perfomances par rapport à un noyau linéaire.



Nous obtenons les erreurs suivantes:

	Erreur	Ecart type
vanilla	0.0081008	0.0040398
poly	0.0339034	0.0094051
rbf	0.0159016	0.0088564

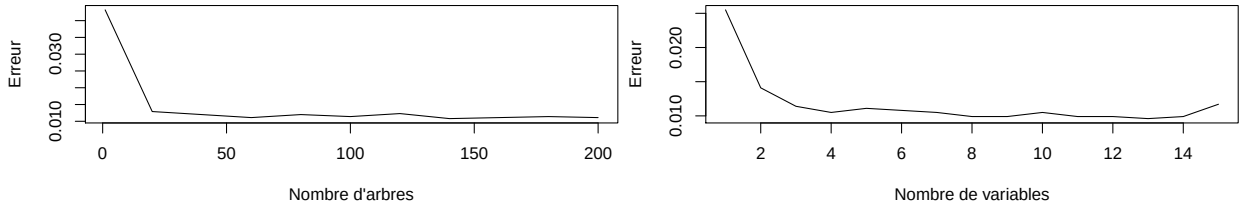
Forêts aléatoires

Nous essayons maintenant les forêts aléatoires.

Nous devons tout d'abord nous assurer que notre fôret aléatoire contient suffisamment d'arbres. En effet, si le nombre d'arbres est trop faible certains individus ne seront pas utilisés (à cause du bootstrap). Si le nombre d'arbres est élévé la recherche de l'hyperparamètre 'mtry' (que nous faisons ensuite) sera très longue. Pour cela nous calculons l'erreur de classification par validation croisée pour des fôrets ayant entre 1 et 200 arbres (par pas de 20).

Ensuite, nous avons un hyperparamètre à optimiser qui est 'mtry' dans la bibliothèque 'randomForest'. Cet hyperparamètre correspond au nombre de variables sélectionnées, par échantillonnage, comme candidates pour chaque noeud. Plus il y a de variables meilleure sera la séparation des données à chaque noeud mais les arbres seront similaires et donc l'agrégation des arbres améliorera peu les performances (par rapport à un arbre seul).

Le nombre de variables minimisant l'erreur de classification est 9.



Nous obtenons l'erreur suivante:

Erreur	Ecart.type
0.0108011	0.005559

Modèle retenu

Le modèle retenu est le SVM avec un noyau linéaire puisqu'il correspond au modèle ayant la plus faible erreur de classification. Nous évaluons l'erreur sur l'ensemble de test (réservé à cet effet au départ): cette erreur nous donne une estimation non biaisée de l'erreur de classification.

```
astr.mod <- ksvm(y~., data=astronomy_train, type="C-svc", kernel='vanilla', C=3125, cross=0)
```

```
##  Setting default kernel parameters
```

```
save(centers, scales, astr.mod, file="env_astronomy.RData")
```

```
rm(list=setdiff(ls(), "astronomy_test"))
astronomy_test_y <- astronomy_test$y
astronomy_test_X <- astronomy_test
astronomy_test_X$y <- NULL
source(file="predicteurs.R")
y_pred <- classifieur_astronomie(astronomy_test_X)
```

Nous obtenons la matrice de confusion suivante sur le jeu de test:

	GALAXY	QSO	STAR
GALAXY	838	10	0
QSO	0	138	0
STAR	3	0	678

La précision est de 99.2%

Maïs

Chargement des données et des fonctions

```
source("functions.R")
data <- read.table(file = "data/mais_train.csv", sep=',', header = T)
data$X <- NULL
RECHERCHE_HYPERPARAMETRE <- FALSE
```

Chargement des librairies

```
library(tidyr)
library(dplyr)
library(ggplot2)
library(randomForest)
library(keras)
library(kernlab)
```

Séparation des données en entrainement/test

Comme précédemment, nous séparons notre datset en deux ensembles: un ensemble d'entrainement et un ensemble de test. Nous centrons réduisons également nos données car elles ont des ordres de grandeur trés différents.

Méthode d'évaluation

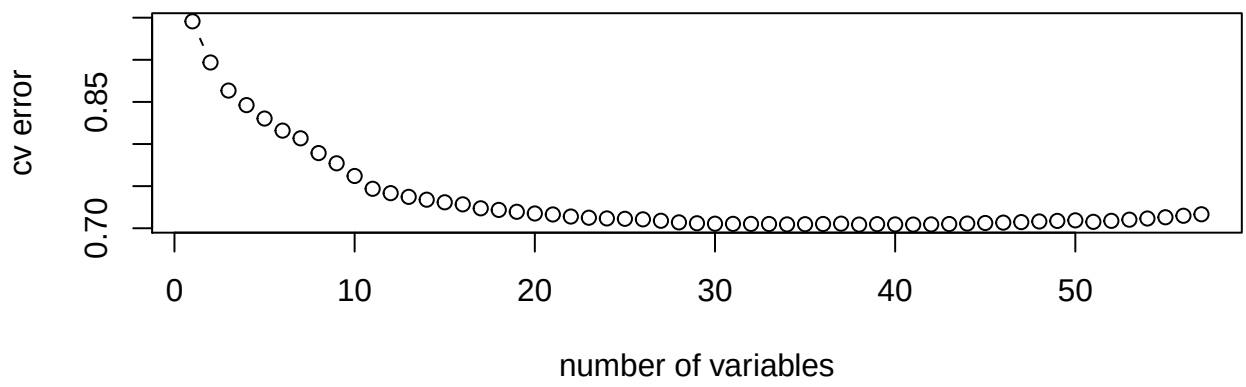
Nous avons choisi l'erreur quadratique moyenne comme mesure de l'erreur, elle sera estimée par validation croisée et par le "out-of-bag error" pour la forêt aléatoire.

Régression linéaire

Nous commençons par une régression linéaire, une méthode qui de par sa simplicité permet une meilleure interprétation.

Nous effectuons une "forward stepwise selection" et pour chaque modèle nous évaluons l'erreur grâce à une k-fold cross-validation. Le meilleur modèle correspond à celui comprenant les 38 premières variables sélectionnées

```
res <- ForwardCrossValidationLM(mais_train)
mais.mod <- lm(as.formula(paste("yield_anomaly ~", paste(res$vars[1:which.min(res$CV)],collapse="+"))),
plot(res$CV, xlab="number of variables", ylab="cv error", type="b")
```



La régression linéaire ainsi que d'autres modèles linéaires tel que la régression Lasso n'arrivent pas à produire de meilleurs résultats. Nous avons opéré des transformations afin d'obtenir un modèle non linéaire en: - multipliant les variables par rapport à elle même - multipliant les variables par rapports aux autres du même mois - transformant la variable qui correspond au département en plusieurs variables binaires afin d'éliminer la relation d'ordre entre les départements (à priori, il n'y a pas de relation linéaire entre le numéro de département et le rendement de maïs) L'erreur quadratique moyenne ne s'est que faiblement amélioré, nous avons par conséquent écarté ses pistes.

Forêts aléatoires

Optimisation du nombre d'arbre

Comme pour le dataset astronomy, nous optimisons dans un premier temps le nombre d'arbre à utiliser en gardant le paramètre mtry fixe (ici p/3). L'erreur est calculé grâce au paramètre "out of bag error" qui est

retourné par la fonction “randomForest”.

```
scaled_test <- data.frame(t(apply(mais_test[,-which(names(mais_test)=="yield_anomaly")], 1, function(r)

nbtree <- optimizeNtreeAndPlotResults()
```

```
## Nombre d'arbres optimal:  956
```

Optimisation de ‘mtry’

```
res <- optimizeMtryAndPlotResults()
```

```
## Param?tre mtry optimal:  34
```

```
test.err <- res$test.err
oob.err <- res$oob.err
mtry <- res$mtry
```

SVM à noyau

Noyau linéaire

Comme dit dans l’exercice précédent, pour des raisons de temps calculs nous choisirons d’optimiser le paramètre C uniquement avec le noyau linéaire.

Nous faisons varier ce paramètre et le calcul de l’erreur se fait par validation croisée

```
if(RECHERCHE_HYPERPARAMETRE){
  C <- (1:5)^5
  CV1 <- numeric(length(C)) # [1] 0.7318539 0.7338461 0.7423543 0.8082920 1.0046964

  for (c in 1:length(C)) {
    svmfit <- ksvm(yield_anomaly ~ .,data = mais_train, scaled=TRUE, type="eps-svr",
    kernel="vanilladot", C=C[c], cross=10)
    CV1[c] <- cross(svmfit)
  }
  c <- C[which.min(CV1)] #1
} else {
  c <- 1
}
```

Le paramètre C optimal trouvé vaut 1 et l’erreur est d’environ 0.73.

Noyau gaussion

Nous faisons de même ici avec le paramètre sigma

```
if(RECHERCHE_HYPERPARAMETRE) {
  Sigma <- seq(0.0001, 0.0401, by=0.005)
  CV <- numeric(length(Sigma)) # [1] 0.8649978 0.6319195 0.5883565 0.5764376 0.5767431 0.5762012 0.5819

  for(s in 1:length(Sigma)) {
    svmfit <- ksvm(yield_anomaly ~ .,data = mais_train, scaled=TRUE, type="eps-svr",
```

```
    kernel="rbfdot", kpar=list(sigma=Sigma[s]), C=c, cross=10)
    CV[s] <- cross(svmfit)
  }
  s <- Sigma[which.min(CV)]
} else {
  s <- 0.0351
}
```

La valeur optimal trouvée 0.0351 et l'erreur estimée vaut 0.57

Noyau polynomial

```
if(RECHERCHE_HYPERPARAMETRE) {
  degree <- 2:5
  CV <- numeric(length(degree)) # [1] 3.226894 1.162719 1.394694 2.193799
  for(d in 1:length(degree)) {
    svmfit <- ksvm(yield_anomaly ~ .,data = mais_train, scaled=TRUE, type="eps-svr",
    kernel="polydot", kpar=list(degree=degree[d]), C=c, cross=10)
    CV[d] <- cross(svmfit)
  }
  d <- degree[which.min(CV)] # 3
} else {
  d <- 3
}
```

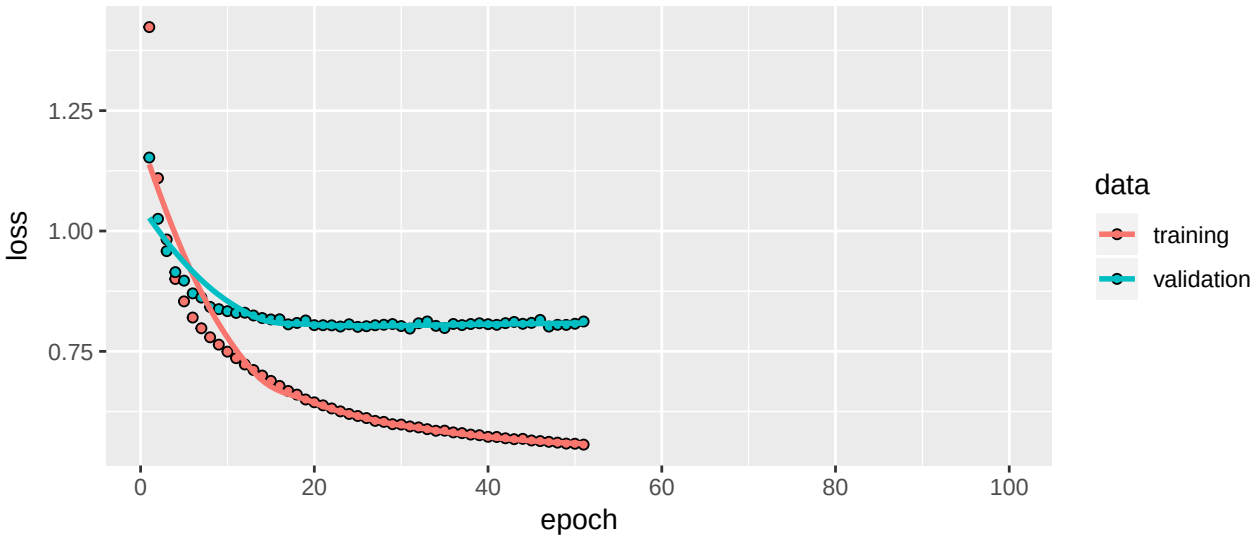
Le noyau polynomial n'offre quant à lui, pas un bon niveau de performance.

Réseau de neurone : construction d'un perceptron

Nous construisons un perceptron avec une couche cachée et 5 neurones dans cette dernière. Avec un nombre de neurones supérieur, notre réseau "surapprend" nos données d'entrainement. Il nous faudrait plus de données pour palier à ce problème étant donné le nombre élevé de variables.

```
history <- build_perceptron()
```

```
plot(history)
```



L'erreur se trouve aux alentours 0.75. Afin de réduire la dimension de notre dataset, nous avons tenté d'utiliser seulement les variables les plus importantes de la forêt aléatoire, nous avons obtenu un taux d'erreur similaire en utilisant plus de neurones.

Modèle retenu

Le modèle retenu est le SVM avec un noyau gaussien, il semble posséder la plus faible erreur même si la forêt aléatoire posséde également de bonnes performances. Nous évaluons également l'erreur sur l'ensemble de test.

```
mais.mod <- ksvm(yield_anomaly ~ .,data = mais_train, scaled=TRUE, type="eps-svr", kernel="rbfdot", kpa
save(oob.err,test.err,centers, scales, mais.mod, file="env_mais.RData")
```

```
source(file="predicteurs.R")
pred <- classifieur_ble(mais_test[,-which(names(mais_test)=="yield_anomaly")])
mse <- mean((pred - mais_test$yield_anomaly)^2)
```

Nous obtenons une erreur d'environ 0.52 sur notre ensemble de test.

```
library(keras)
library(jpeg)
```

Les données

Ce problème est un problème de classification. Chaque image contient un objet qui est sa classe. Nous avons trois classes: voitures, chats, et fleurs. Il y a au total 1596 images.

Chargement des images

```
filenames_car <- array(list.files(path='data/images_train/car', full.names = TRUE, pattern='*.jpg'))
filenames_cat <- array(list.files(path='data/images_train/cat', full.names = TRUE, pattern='*.jpg'))
filenames_flower <- array(list.files(path='data/images_train/flower', full.names = TRUE, pattern='*.jpg
```

Nous obtenons la répartition suivante selon les classes. Il y a à peu près autant d'individus dans chaque classe.

	Nombre d'images
Car	485
Cat	590
Flower	521

Création des étiquettes

Nous créons un vecteur contenant les étiquettes des données: 0 pour les voitures, 1 pour les chats, 2 pour les fleurs.

```
y <- c(rep(0, length(filenames_car)), rep(1, length(filenames_cat)), rep(2, length(filenames_flower)))
```

Séparation des données d'entrainement, de validation et de test

Nous séparons les données en trois ensembles: entrainement valdiation et test. L'ensemble de validation va nous servir à faire un arrêt précoce: lorsque le réseau de neurones n'améliorera plus la précision sur l'ensemble de validation l'entrainement s'arrêtera. Cela permet de limiter le surentrainement. L'ensemble de test serivara à caluler une estimation de l'erreur de classification non biaisée.

```
set.seed(123)

filenames <- c(filenames_car, filenames_cat, filenames_flower)

idx <- sample(seq(1, 3), size = length(filenames), replace = TRUE, prob = c(.6, .2, .2))
filenames_train <- filenames[idx == 1]
filenames_val <- filenames[idx == 2]
filenames_test <- filenames[idx == 3]
y_train <- y[idx == 1]
y_val <- y[idx == 2]
y_test <- y[idx == 3]
```

Transformation des données

Nous utilisons les pixels comme variables d'entrée et donc nous n'avons pas besoin d'extraire des caractéristiques de l'image. Cependant toutes les images n'ont pas la même taille nous réalisons donc une interpolation bilinéaire pour que toutes les images aient la même taille. Nous avons choisi une taille de 100x100.

```
dim_images <- c(100, 100)
images_train <- read_images(filenames_train, dim_images)
images_val <- read_images(filenames_val, dim_images)
```

Modèle

Notre modèle est un réseau de neurones convolutif. Ce type de modèle est adapté aux images puisque de part sa construction il prend en compte l'invariance aux transaltions des images.

Les images en entrées ont les dimensions: 100x100x3 (Hauteur x Largeur x Nombre de couleurs). Ensuite nous appliquons des convolutions jusqu'à obtenir un vecteur de taille 2x2x64 que nous "applatissons" en un vecteur de taille 256. Chaque convolution a un filtre de taille 3x3 (pour chaque carte de caractéristique en entrée et en sortie). Nous choisissons de réduire la hauteur et la largeur des cartes de caractéristiques

uniquement avec le pas des convolution (donc nous n'utilisons pas de pooling et mettons systématiquement du padding). Chaque convolution est suivi d'uneactivation 'relu' pour mettre de la non-linéarité. Le nombre de carte de caractéristiques est augmenté progressivement de 3 cartes (pour les 3 couleurs) jusqu'à 64 de façpn à extraire de plus en plus d'informations au fur et à mesure.

Ensuite nous passons le vecteur de taille 256 dans 2 couches entièrement connectées. La dernière couche entièrement connectée est suivie d'un softmax. Le modèle est entrainé par minimisation de l'entropie croisée ce qui revient à la maximisation du maximum de vraisemblance de notre modèle.

Afin d'éviter le surentrainement nous utilisonsdu dropout avec une probabilité de 0.5 (chaque neurone a une chance sur deux de renvoyer zéro). De plus nous effectuons un arrêt précoce lorsque le CNN n'a pas progressé lors de 10 epochs.

Notre modèle contient 185 000 paramètres et atteint 92% de précision sur l'ensemble de validation.

```
model <- keras_model_sequential()

model %>%
   layer_conv_2d(
    filter=16,kernel_size=c(3,3),padding="same",strides=c(2, 2), input_shape = c(dim_images[1], dim_ima
  ) %>%
  layer_conv_2d(filter=16,kernel_size=c(3,3),padding="same",activation="relu") %>%

  layer_conv_2d(filter=16,kernel_size=c(3,3),padding="same",strides=c(2, 2),activation="relu") %>%
  layer_conv_2d(filter=16,kernel_size=c(3,3),padding="same",activation="relu") %>%
  layer_dropout(0.5) %>%

  layer_conv_2d(filter=32,kernel_size=c(3,3),padding="same",strides=c(2, 2),activation="relu") %>%
  layer_conv_2d(filter=32,kernel_size=c(3,3),padding="same",activation="relu") %>%

  layer_conv_2d(filter=32,kernel_size=c(3,3),padding="same",strides=c(2, 2),activation="relu") %>%
  layer_conv_2d(filter=32,kernel_size=c(3,3),padding="same",activation="relu") %>%
  layer_dropout(0.5) %>%

  layer_conv_2d(filter=64,kernel_size=c(3,3),padding="same",strides=c(2, 2),activation="relu") %>%
  layer_conv_2d(filter=64,kernel_size=c(3,3),padding="same",activation="relu") %>%

  layer_conv_2d(filter=64,kernel_size=c(3,3),padding="same",strides=c(2, 2),activation="relu") %>%
  layer_conv_2d(filter=64,kernel_size=c(3,3),padding="same",activation="relu") %>%

  layer_flatten() %>%
  layer_dropout(0.5) %>%

  layer_dense(64,activation="relu") %>%
  layer_dropout(0.5) %>%

  layer_dense(3,activation="softmax")

opt <- optimizer_adam(lr = 0.001)

model %>% compile(loss="sparse_categorical_crossentropy",optimizer=opt,metrics="accuracy")
```

Entrainement

Nous entrainons notre modèle avec des lots de 30.

```
early_stopping <- callback_early_stopping(monitor = 'val_acc', patience = 10)
history <- model %>% fit(images_train,y_train,batch_size=30,epochs=60,
                         validation_data=list(images_val, y_val),shuffle=TRUE,
                         callbacks=c(early_stopping),verbose=0)
```

Evaluation de l'erreur sur l'ensemble de test

Nous évaluons l'erreur sur l'ensemble de test (réservé à cet effet au départ): cette erreur nous donne une estimation non biaisée de l'erreur de classification.

```
model %>% save_model_hdf5("model.h5")
```

```
rm(list=setdiff(ls(), c("filenames_test", "y_test")))
source(file="predicteurs.R")
y_pred <- classifieur_images(filenames_test)
```

Nous obtenons la matrice de confusion suivante sur le jeu de test:

	car	cat	flower
car	93	1	4
cat	3	106	10
flower	1	4	83

La précision est de 92% sur l'ensemble de test.