TP : Apprentissage Statistique

22 et 24 novembre 2023

Ce document constitue une suite de 9 activités pratiques à réaliser avec le langage R pour passer progressivement de l'apprentissage *supervisé* à quelques notions regardant l'apprentissage dit *non-supervisé*, c'est-à-dire lorsque le jeu d'entraînement est exclusivement composé de données non-étiquetées. Les questions qui peuvent être posées dans un tel cadre sont en général plus diverses : partitionner les données en sous-groupes, en trouver un modèle génératif, réduire leur dimensionnalité, créer de nouveaux descripteurs plus expressifs, représenter des données abstraites dans un espace vectoriel de dimension choisie *a priori*, détecter des anomalies... Notons que les deux approches ne sont en général pas mutuellement exclusives : il arrive bien souvent qu'une approche non-supervisée permette de préparer le terrain pour une implémentation efficace d'une approche supervisée [1].

Nous partirons du cadre confortable de l'apprentissage supervisé, pour retirer progressivement des informations - classes, étiquettes puis descripteurs - jusqu'à arriver au cas de données abstraites d'un espace métrique.

1. Introduction... *Premiers pas en apprentissage supervisé*

Dans cette première activité, nous allons utiliser trois modèles d'apprentissage :

- le modèle bayésien naïf (*Naive Bayes*)
- un arbre de décision (*Classification and Regression Tree* ou CART)
- une forêt d'arbres aléatoires (*Random Forest*)

Nous allons utiliser deux jeux de données :

- `heart.csv` : dans un problème de ***classification*** visant à prédire des insuffisances cardiaques à partir de 11 paramètres cliniques observés sur 918 patients.

- `diamonds.csv` : dans un problème de ***régression*** visant à prédire la valeur monétaire (en $) de diamants en fonction de leurs dimensions et de leurs paramètres physiques (9 paramètres). Le jeu de données comporte 53 914 instances.

Partie A. Classification

Avant de commencer, nous devons installer la bibliothèque `e1071` permettant d'utiliser le classifieur bayésien naïf :

```
install.packages("e1071")
```

On pourra alors charger la bibliothèque `e1071` dans l'entête du script :

```
library('e1071')
```

Enfin, charger le jeu de données :

1. Le cas de figure inverse est plus rare mais peut aussi se rencontrer.

```
data = read.csv("datasets/tp1/heart.csv")
```

Q1. Séparer le jeu de données en deux sous-ensembles : `train` et `test`, contenant chacun la moitié de l'effectif. Pour assurer une répartition aléatoire de la variable cible dans les deux jeux, on mettra toutes les données d'indice impair dans le jeu `train`, et toutes les données d'indice pair dans le jeu `test`.

Q2. Pour entraîner un modèle de classifieur bayésien naïf, on utilisera la fonction `naiveBayes` de la bibliothèqe `e1071` :

```
model = naiveBayes(Y ~ ., data = dataset)
```

où `Y` est la variable cible du problème et où `dataset` est le nom du jeu de données utilisé pour l'entraînement. La syntaxe `Y` $\sim$. indique que l'on cherche à prédire la variable `Y` à partir de toutes les autres variables du *data frame* `dataset`.

Entraîner un classifieur bayésien naïf sur le jeu `train`, pour apprendre à prédire la variable d'insuffisance cardiaque `HeartDisease` à partir des paramètres cliniques observés. Inspecter le contenu de la variable `model`.

Q4. Pour appliquer le modèle sur un jeu de données de descripteurs dont on souhaite prédire les variables cibles, on utilise la fonction `predict` :

```
yhat = predict(model, newdata = dataset))
```

où `dataset` est le nom du jeu de données utilisé pour la prédiction.

Appliquer le modèle en prédiction sur le jeu de données `test`.

Remarque : ces deux syntaxes - pour l'entraînement (Q3.) et la validation (Q4.) - sont communes à la grande majorité des modèles d'apprentissage automatique en R, ce qui rend très aisé la comparaison des performances de plusieurs modèles sur un même problème.

Q5. L'évaluation des performances d'un classifieur se fait simplement en calculant une table de contingence entre les variables cibles y (`HeartDisease`) et les variables prédites $\widehat{y}$ (`yhat`).

```
T = table(y, yhat)
```

Inspecter le contenu de la table `T`.

Calculer le taux de classification correcte, *i.e.* le taux d'instances correctement classifiées par rapport à la vérité terrain.

Nous allons à présent tester un second modèle d'apprentissage .

Avant de commencer, nous devons installer les bibliothèque `rpart` et `rpart.plot` permettant d'utiliser les fonctionnalités des arbres de décision :

```
install.packages("rpart")
install.packages("rpart.plot")
```

Charger ces bibliothèques dans l'entête du script :

```
library('rpart')
library('rpart.plot')
```

Q6. On utilise la séparation en `train` et `test` effectuée dans la question Q1.

La création d'un modèle d'arbre de décision nécessite de spécifier les paramètres de l'algorithme dans une variable `ctrl`. Ici, nous allons seulement spécifier la profondeur maximal de l'arbre.

```
ctrl = rpart.control(maxdepth=6)
```

Pour créer un modèle, on utilise la fonction `rpart` :

```
model = rpart(Y ~ ., data = dataset, control=ctrl)
```

où `Y` est la variable cible du problème et où `dataset` est le nom du jeu de données utilisé pour l'entraînement. La syntaxe `Y` $\sim$. indique que l'on cherche à prédire la variable `Y` à partir de toutes les autres variables du *data frame* `dataset`.

Q7. Pour appliquer le modèle sur un jeu de données dont on souhaite prédire les variables cibles, on utilise l'instruction :

```
yhat = predict(model, newdata = dataset))
```

où `dataset` est le nom du jeu de données utilisé pour la prédiction.

Appliquer le modèle en prédiction sur le jeu de données `test`.

Q8. Observer et interpréter le résultat de la prédiction, et convertir le vecteur `yhat` en une variable décisionnelle valant `1` si le modèle prédit que le sujet présente une insuffisance cardiaque et `0` sinon.

Q9. Avec une démarche similaire à celle de la question Q5., calculer le taux de classification correcte du modèle.

Partie B. Régression

Avant de commencer, nous devons installer la bibliothèque `randomForest` :

```
install.packages("randomForest")
```

Charger cette bibliothèque dans l'entête du script :

```
library('randomForest')
```

Q10. Charger et inspecter les données du fichier `diamonds.csv`.

Q11. Après avoir séparé le jeu de données en deux parties `train` et `test`, construire un modèle de régression pour prédire la variable `price` à partir de toutes les autres variables du jeu de données.

Q12. Pour valider le modèle, calculer la racine carré de l'erreur quadratique moyenne (RMSE) de la variable prédite $\widehat{y}$ par rapport à la valeur vraie y :

$$\text{RMSE} = \sqrt{\mathbb{E}\big[(\widehat{y} - y)^2\big]}$$

Quelle est la part de variance expliquée par le modèle ?

2. Notion de baseline... *Validation opérationnelle d'un modèle d'apprentissage*

Dans cette activité, nous allons utiliser le jeu de données `semis.csv` composé de 10 000 relevés forestiers. Chaque relevé est une observation ponctuelle, pour laquelle on dispose des observables suivants :

- `x`, `y` : coordonnées spatiales du relevé, dans le système RGF 93 - projection Lambert 93
- `nature` : essence forestière principale au voisinage du relevé
- `perimetre` : en m
- `surface` : en m^2
- `publique` : le caractère public ou non de la zone echantillonnée (valeur booléenne)
- `distNNhab` : distance à l'habitation la plus proche (en m)
- `distNNrout` : distance au réseau routier, hors sentiers (en m)
- `distNNhydr` : distance au réseau hydrographique (en m)
- `alt_1` : altitude du terrain, système NGF-IGN 69 (en m)
- `pente_1` : pente du terrain (en °)

De plus, pour chaque instance de donnée, l'attribut `ancienne` (à valeurs booléennes) nous dit si le relevé a été effectué dans une parcelle forestière ancienne [2] ou récente.

Q1. Charger les données du fichier `semis.csv` dans l'environnement R.

```
data = read.csv("datasets/tp2/semis.csv")
```

Q2. Séparer le jeu de données en deux sous-ensembles : `train` et `test`, contenant chacun la moitié de l'effectif. On mettra toutes les données d'indice impair dans le jeu `train`, et toutes les données d'indice pair dans le jeu `test`.

Q3. À l'aide du modèle d'apprentissage de votre choix (Naive Bayes, arbre de décision ou forêt d'arbres aléatoires), construire à partir du jeu `train` un classifieur permettant de déterminer si un relevé forestier est issu d'une parcelle de forêt ancienne ou récente.

Q4. Valider le modèle sur le jeu `test` et évaluer le taux de classification correcte.

Q5. Discuter de la pertinence du résultat obtenu.

Q6. Charger à présent le fichier `semis2.csv` sur lequel on appliquera les opérations décrites dans les questions Q2 à Q5.

Q7. Proposer et mettre en oeuvre une piste d'amélioration du protocole d'apprentissage.

2. Dans ce contexte, on considère comme ancienne, une parcelle de forêt déjà signalée sur les cartes d'Etat-Major (XIX^e).

3. Forêts aléatoires... *en contexte opérationnel*

L'objectif de cette activité est d'apprendre à utiliser le modèle des forêts aléatoires en classification. Pour ce faire nous utilisons le jeu de données `chateau.wkt`, pour lequel nous chercherons à prédire si un bâtiment vecteur issu de la base de données topographique de l'IGN est un château (problème de classification).

Cette activité nécessite (si ce n'est déjà fait) l'installation de la bibliothèque `randomForest` :

```
install.packages("randomForest")
```

Charger cette bibliothèque dans l'entête du script :

```
library('randomForest')
```

Q1. Charger le jeu de données `chateau.wkt` et récupérer les colonnes correspondant aux descripteurs et à la variable cible.

```
chateau = read.csv("datasets/tp3/chateau.wkt", sep=";")
chateau = chateau[,4:25]
chateau$ischateau = as.factor(chateau$ischateau)
```

La troisième instruction ci-dessus permet de convertir la colonne cible `ischateau` en facteur (étape nécessaire avec la bibliothèque `randomForest`, pour indiquer que la variable cible est de type catégorielle).

On pourra représenter les données vecteurs de bâtiments dans le logiciel QGIS.

Q2. Séparer le jeu de données en deux sous-ensembles : `train` et `test`, contenant chacun la moitié de l'effectif. Pour assurer une répartition aléatoire de la variable cible dans les deux jeux, on mettra toutes les données d'indice impair dans le jeu `train`, et toutes les données d'indice pair dans le jeu `test`.

Q3. Entraîner un modèle de forêt aléatoires sur le jeu `train` pour apprendre à prédire la variable `ischateau`. On prendra un modèle composé de 500 arbres.

Q4. Calculer la courbe ROC du modèle. On pourra utiliser directement le script suivant (source : `https://www.r-bloggers.com/2016/08/roc-curves-in-two-lines-of-r-code/`) :

```
roc = function(labels, scores){
  labels = labels[order(scores, decreasing=TRUE)]
  return(data.frame(TPR=cumsum(labels)/sum(labels), FPR=cumsum(!labels)/sum(!labels), labels))
}
```

La fonction `roc` prend en entrée un vecteur `labels` de variables cibles *réelles* (composé de 0 et de 1), et un vecteur `scores` contenant les probabilités $P(Y_i = 1|\mathbf{X_i})$ pour chaque donnée à valider.

Indication : pour pouvoir calculer la courbe ROC, lors de l'étape de prédiction on doit préciser que l'on souhaite un retour sous forme de probabilités, en précisant `type="prob"` dans les arguments de `predict`, puis en récupérant, sous forme numérique (`as.numeric`) la deuxième colonne qui correspond à la probabilité $P(Y_i = 1|\mathbf{X_i})$, la première colonne correspondant à la probabilité complémentaire $P(Y_i = 0|\mathbf{X_i})$:

```
proba = as.numeric(predict(model, newdata, type="prob")[,2])
```

Pour aller plus loin...

Q5. Calculer l'aire sous la courbe (AUC) du modèle.

Q6. En utilisant la technique du *bootstrap statistique* évaluer graphiquement une bande de confiance autour de la courbe ROC du modèle. Pour ce faire, on génèrera 30 échantillons bootstrap du jeu de données `train`. Pour chaque échantillon, on évaluera une nouvelle courbe ROC, que l'on superposera sur le tracé graphique de la courbe principale calculée à la question Q4.

Q7. En utilisant à nouveau le bootstrap, calculer un intervalle de confiance à 95% autour de la valeur estimée de l'AUC.

Q8. Analyser l'importance des variables explicatives.

Q9. Relancer l'entraînement d'une forêt aléatoire sur le jeu `train`, en choisissant 50 arbres et avec le paramètre `do.trace=1`. Analyser la sortie console de l'algorithme. Que signifie la colonne OOB ?

4. Champignons... *Un TP de synthèse sur l'apprentissage supervisé*

Dans ce TP, nous utiliserons le jeu de données `mushroom.dat`[3] répertoriant les caractéristiques (pour la plupart catégorielles) de 1625 espèces de champignon. On s'intéresse à la comestibilité de chaque espèce, renseignée dans la variable binaire `class`, comportant les modalités `'e'` (*edible*) et `'p'` (*poisonous*).

L'objectif du TP consiste à prédire cette comestibilité à l'aide des caractéristiques phénotypiques des espèces. Nous comparerons deux algorithmes d'apprentissage : les **arbres de décision** et les **forêts aléatoires**.

Pour ce faire, nous utiliserons deux librairies, à charger dans votre script (après installation si ce n'est déjà fait, avec la commande `install.packages`).

```
library("rpart")
library("randomForest")
```

Pour garantir la reproductibilité exacte des résultats, il est fortement conseillé d'utiliser la graine aléatoire suivante :

```
set.seed(12345)
```

Q1. Charger et inspecter les données du fichier `mushroom.dat`.

Q2. Pour ce TP, afin de garantir un volume de données suffisant dans la construction du modèle d'apprentissage, nous opérerons une séparation 80-20. Subdiviser le jeu de données complet pour créer un *data frame* `train` contenant 80% des données, et un *data frame* `test` contenant les 20% restants. La subdivision sera effectuée de manière régulière, en plaçant alternativement et régulièrement 4 lignes du 5 dans le jeu `train`, puis la 5eme ligne dans `test`.

Q3. Sur les données d'entraînement, construire deux modèles d'apprentissage :

- un arbre de décision (package `rpart`)
- une forêt aléatoire (package `randomForest`)

Quel est le taux d'erreur du modèle prédit par l'échantillon *Out of Bag* de la forêt aléatoire ? Interprétation ?

Q4. Valider les deux modèles ainsi construits sur le jeu de test. Indiquer leur sensibilité, leur spécificité, leur précision (PPV) et leur mesure F_1. Lequel de ces deux classifieurs semble le plus performant ?

Q5. À l'aide d'un calcul théorique, évaluer grossièrement la marge d'erreur sur les indicateurs calculés dans la question Q4. La conclusion tirée est-elle toujours aussi certaine ?

Q6. Calculer et représenter (sur le même graphique) les courbes ROC de chacun des deux modèles. Pour ce faire, on pourra utiliser le code de calcul épuré suivant :

```
roc = function(labels, scores){
       labels = labels[order(scores, decreasing=TRUE)]
       return(data.frame(FPR=cumsum(!labels)/sum(!labels), TPR=cumsum(labels)/sum(labels)))
}
```

Q7. À l'aide du code ci-dessous, calculer l'aire sous la courbe (AUC) de chacune des deux courbes ROC. La différence entre les deux classifieurs semble-t-elle significative ? On rappelle que l'AUC est un indicateur de qualité global d'un classifieur binaire, exprimé en pourcentage : une AUC de 100 % indique un classifieur parfait, tandis qu'une AUC de 50 % indique un classifieur aléatoire pur.

3. Extrait (partiellement) du site OpenML : `https://www.openml.org/d/24`

```
auc = function(ROC){
    N = nrow(ROC)
    dFPR = ROC[2:N,"FPR"]-ROC[1:(N-1),"FPR"]
    return(sum(dFPR*ROC[2:N,"TPR"]))
}
```

Q8. Pour confirmer l'intuition développée à la question Q5, opérer un bootstrap statistique sur l'échantillon de validation pour générer (et représenter) 100 réplications de la courbe ROC de chacun des deux classifieurs. Calculer la p-value de l'hypothèse nulle H_0 : *le aires sous la courbe (AUC) des deux modèles sont égales*, et interpréter le résultat obtenu.

Q9. Reprendre les questions Q7 et Q8 en remplaçant l'échantillon de validation (contenant 20% de l'effectif total du jeu de données) en implémentant un système de validation croisée : diviser le jeu de données total en 5 parties, puis utiliser chacune de ces parties tour à tour comme jeu de validation, les 4 autres parties étant utilisées pour l'entraînement. À l'issue de cette phase, on doit pouvoir produire une validation sur la totalité du jeu de données. En utilisant à nouveau le bootstrap statistique, montrer que les courbes ROC (et les AUC correspondantes) sont suffisamment précises pour conclure quant à la superiorité du modèle des forêts aléatoires sur ce problème d'apprentissage.

Q10. Pour une application concrète, on souhaite que le risque d'empoisonnement soit inférieur à 1 pour 1000. Quelle proportion de champignons comestibles peut-on espérer collecter dans le meilleur des cas ?

5. Apprentissage structuré... *Ou quand la cible du problème n'est pas clairement identifiée.*

Pour cette activité, nous allons avoir besoin du package `bnlearn` spécifiquement conçu pour manipuler des modèles graphiques probabilistes (*cf* cours de Statistique & Probabilités, slides 62-65) :

```
install.packages("bnlearn")
```

Optionnellement, on pourra aussi installer le package `Rgraphviz` pour la représentation des modèles sous forme de graphes topologiques :

```
install.packages("BiocManager")
BiocManager::install("Rgraphviz")
```

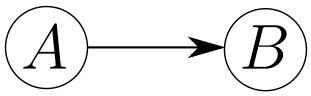
Si l'installation a été effectuée sans erreurs, charger ces deux librairies dans l'entête d'un script R :

```
library("bnlearn")
library("Rgraphviz")
```

Q1. Échauffement. Les performances d'un test antigénique sont évaluées respectivement à 75 % de sensibilité et 0.5 % de risque de faux positif. On suppose qu'à une date donnée, le taux de positivité au SARS-COV-2 (au sein d'une population de personnes testées) est de 8 %. Sachant que le test d'un patient est positif, évaluer la probabilité que ce dernier soit réellement malade.

Réévaluer cette probabilité pour un taux de positivité de 30 %.

Q2. On modélise le problème informatiquement à l'aide d'un graphe à 2 noeuds : A pour l'état du patient (deux modalités : *sain* ou *malade*) et B pour le résultat du test (deux modalités : *négatif* et *positif*).



On rappelle que la loi jointe associée à ce modèle est donnée par la forme factorisée (règle de chaînage) :

$$P(A, B) = P(A)P(B|A)$$

Créer le modèle sous R avec la commande suivante :

```
G = model2network("[A][B|A]")
```

On pourra tracer le modèle avec la commande `plot(G)`.

On va ensuite encoder les paramètres du modèle (*i.e.* les paramètres de la loi *a priori* $P(A)$ et de la loi conditionnelle $P(B|A)$). Pour ce faire, on utilise les commandes suivantes :

```
modA = c("SAIN", "MALADE"); modB = c("NEG", "POS")
pA = matrix(c(0.92, 0.08), ncol = 2, dimnames = list(NULL, modA))
pB = matrix(c(0.995, 0.005, 0.25, 0.75), ncol = 2, dimnames = list("B" = modB, "A" = modA))
model = custom.fit(G, dist = list(A = pA, B = pB))
```

Analyser ces trois lignes de code pour en comprendre la logique sous-jacente.

Muni de ses paramètres, le modèle peut à présent être représenté plus esthétiquement avec le package `Rgraphviz` :

```
graphviz.plot(model, layout = "dot")
```

La différence entre les deux représentations paraît pour l'instant dérisoire, mais elle aura toute son importance ultérieurement pour des modèles ayant un plus grand nombre de variables.

Afficher le modèle dans la console R (en tapant simplement `model`) et analyser la sortie produite.

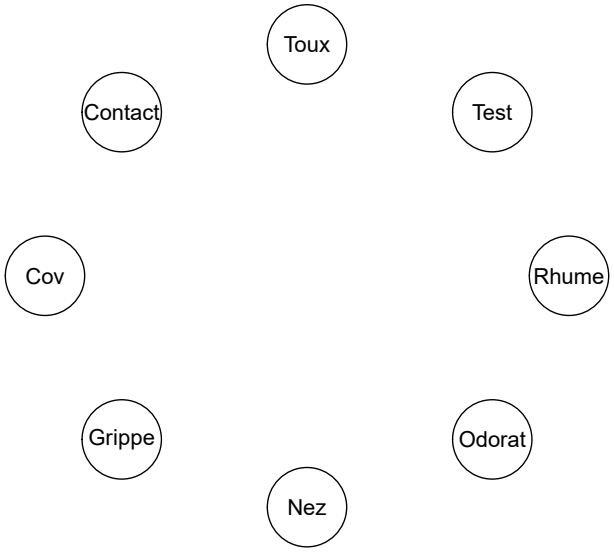
Q3. Pour répondre informatiquement à la question étudiée en Q1, on utilise l'outil `cpquery`, permettant de *requêter* le graphe pour trouver la probabilité que des évènements concernant certains noeuds du graphe se réalisent (argument `event` ci-dessous), étant données des observations sur d'autres noeuds (`evidence`).

```
yhat = cpquery(model, event, evidence, n)
```

Chacun des deux arguments `event` et `evidence` sera encodé sous la forme `(Nom-du-noeud == "modalité")`. Exemple : l'évènement *le patient est sain* sera noté `(A == "SAIN")` (attention à ne pas oublier les parenthèses). Pour combiner plusieurs évènements, on utilise les caractères `|` (union) et `&` (intersection). Par ailleurs, le résultat recherché étant calculé par des méthodes stochastiques, le paramètre `n` indique le nombre de simulations à réaliser. Pour garantir la stabilité des résultats, on pourra fixer une valeur arbitrairement grande, par exemple `n=1e6`.

Quelle est la probabilité que le patient soit réellement malade sachant que son test est positif ? Modifier le réseau pour répondre à la même question pour un taux de positivité de 30 %.

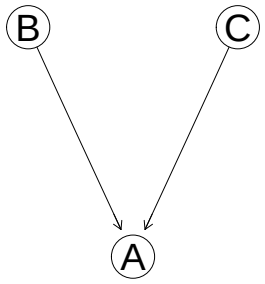
Q4. On considère à présent un modèle à 8 variables binaires : `Covid` (être malade), `Test` (avoir un résultat positif au test antigénique), `Contact` (être considéré comme cas contact au moment du test), auxquelles on ajoute deux affections concurrentes : `Rhume` et `Grippe`, ainsi que trois symptômes communs potentiels : `Toux`, `Nez` (qui coule) et perte d'`Odorat`.



Compléter le graphique ci-après pour ajouter les liens de causalité. Ecrire la forme factorisée de la loi de probabilité jointe correspondante. Combien de paramètres sont nécessaires à la description de cette loi ?

Q5. Nous allons encoder en R le modèle défini dans la question précédente. Utiliser la fonction `model2network` (question Q2) pour créer le modèle. Lorsqu'une variable A est conditionnée par plus d'une variable (*e.g.* B et C), la syntaxe à utiliser est la suivante : `[A|B:C]`. Par exemple, la commande `model2network("[B][C][A|B:C]")` correspond au modèle :

Tracer le modèle avec `plot`.



Q6. Pour estimer les paramètres du modèle, nous allons utiliser un jeu de données d'entraînement. Charger et inspecter le fichier `sars-cov-2.dat` :

```
train = read.csv("datasets/tp5/sars-cov-2.dat")
```

Pour procéder à l'inférence, on utilisera la fonction `bn.fit` prenant en entrée le modèle graphique `G` et les données d'entraînement `train`.

```
model = bn.fit(G, train)
```

Inspecter le modèle obtenu. Tracer le modèle avec `graphviz.plot`.

Q7. En reprenant les indications données à la question Q3, répondre aux questions suivantes :

- Quelle est la probabilité d'être malade sachant que :
  - le patient est cas contact, a une perte d'odorat et de la toux
  - le test est négatif, le patient est cas contact, a une perte d'odorat et de la toux
  - le patient a de la toux
  - le patient a de la toux et le nez qui coule
- Quelle est la probabilité que le résultat du test antigénique d'un patient présentant des signes d'anosmie soit positif ?

Interpréter les résultats numériques obtenus.

Q8. Dans certains cas, il n'existe pas de forme topologique *a priori* pour le modèle graphique. Une solution (à manier avec force précautions !) consiste à apprendre la structure du graphe à partir du jeu d'entraînement. Pour ce faire, il existe de nombreux algorithmes, le plus simple consistant à calculer une matrice de covariance et/ou une table de contingence sur les variables, puis à conserver un lien entre chaque couple de variables pour lequel une dépendance significative est observée. La direction de causalité est alors choisie de sorte à faciliter les calculs d'inférence.

Ici, nous allons utiliser la fonction `iamb` prenant en seule entrée le jeu de données d'entraînement.

Construire automatiquement et tracer le graphe. Comparer la structure obtenue avec celle définie manuellement à la question Q4.

Pour aller plus loin...

On pourra évaluer rigoureusement les performances du modèle en procédant à une séparation entraînement-validation du jeu de données, et en choisissant tour-à-tour plusieurs scénarii (*i.e.* une variable cible, et un ensemble de variables

observées) puis en calculant tous les indicateurs souhaités (sensibilité, spécificité, mesure F_1, courbe ROC, AUC...).

Par ailleurs, lorsque les directions de causalité ne sont pas évidentes (soit qu'elles ne sont pas connues a priori, ou alors que l'influence est mutuelle entre les deux variables concernées), on pourra avoir recours aux modèles graphiques non-dirigés (*Undirected Graphical Models*, ou UGM) tels que les champs de Markov. La propagation du virus entre plusieurs personnes (qui constituent ainsi les noeuds du graphe) en est un cas classique.

Pour plus d'informations, voir le chapitre 3 du polycopié d'apprentissage.

6. Apprentissage de données fonctionnelles... *Ou quand les données sont des courbes.*

Dans cette seconde activité, on s'intéresse au cas où les données sont des objects plus complexes (et plus structurés) qu'une série de descripteurs. C'est le cas en particulier lorsqu'on travaille avec des signaux : audio, vidéo, séries temporelles, données géographiques... Le contenu de l'information n'est alors plus *stricto sensu* la somme des valeurs des descripteurs terme à terme : la même information peut être décrite arbitrairement aussi précisément qu'on le souhaite sous différents formats.

Une manière simple de gérer ce genre d'*inputs* consiste à extraire une série de descripteurs à partir des données brut (*cf* TP apprentissage supervisé : détection de chateaux). En toute rigueur, on souhaite que ces descripteurs ne soient pas sensibles au format d'entrée de l'information, *i.e.* par exemple dans le cas des chateaux, que la valeur des descripteurs calculés ne doit dépendre que du bâtiment et non de sa représentation (raster/vecteur, nombre de points, ordre des sommets, etc). Cependant, il arrive qu'aucun descripteur pertinent ne puisse être extrait manuellement. Les bases fonctionnelles offrent alors un outil de puissant comme nous allons le voir ici.

Charger la librairire `randomForest` et les données du fichier `signals.dat` :

```
library('randomForest')
data = read.csv("datasets/tp6/signals.dat", h=F, sep=" ")
```

Le fichier de données contient 1000 exemple de réalisations de signaux échantillonnés sur 100 points (`V2` à `V101` et classifiés en deux types (variable `V1`). Pour représenter le premier signal, on pourra saisir la commande :

```
plot(as.numeric(data[1,2:101]), type='l')
```

Pour s'assurer que les forêts d'arbres aléatoires travaillent en mode classification, on veillera à transformer la variable de type en facteur, et on supprimera la dernière colonne vide :

```
data$V1 = as.factor(data$V1)
data = data[,1:101]
```

Q1. Après avoir séparé le jeu de données en deux parties égales, construire un modèle de forêt aléatoire sur les données brut, puis valider ce modèle en calculant la table de contingence des prévisions (fonction `table`). Evaluer le taux de classifications correctes. Ce modèle est-il satisfaisant ?

Q2. Nous allons procéder à une transformation de Karhunen-Loève [4] afin d'extraire une base de fonctions adaptée à nos données.

Calculer une ACP des 100 descripteurs sur le jeu de données complet [5]. Pour calculer l'ACP, on pourra utiliser la fonction `prcomp`, prenant en entrée directement le *data frame* de taille 1000×100 (amputé de la première colonne).

Remarquons que le champ `rotation[,i]` de la sortie de `prcomp` contient la `i`-eme fonction de base calculée par l'ACP. Tracer quelques unes de ces fonctions de base. Que remarque-t-on ?

Quelle est la part de variance *capturée* par les 10 premières fonctions de base ?

Q3. Remarquons que le champ `x[i,1:10]` de la sortie de `prcomp` contient les 10 premiers coefficients du `i`-eme signal. Entraîner un nouveau modèle de forêt aléatoire sur ces 10 nouveaux descripteurs et évaluer son taux de classification correcte.

Notons qu'il existe d'autres bases de fonctions utiles pour représenter les données : Legendre, Tchebychev, Fourier, ondelettes, Splines... Pour plus d'informations, voir section 2.3 du polycopié d'apprentissage.

4. Généralisation fonctionnelle de l'analyse en composantes principale (ACP)
5. Rigoureusement, l'ACP devrait être évaluée sur le seul jeu d'entraînement. Pour simplifier le travail, on la calcule ici sur toutes les données. La variable cible `V1` des données de test n'est bien entendu pas utilisée lors de l'entraînement, ce qui justifie la légitimité de la démarche.

7. Détection d'anomalies... *Ou quand on ne possède pas d'exemples positifs*

La détection d'anomalie est un champ de recherche très actif, possédant de nombreux cas d'applications concrets : sécurité, détection de fraude, contrôle qualité, etc. Les anomalies étant par définition très rares, il est en général difficile de disposer d'une base conséquente d'exemples positifs. Nous nous plaçons ici dans le cadre où aucune anomalie n'a été détectée jusqu'à présent, résultant ainsi en une base d'entraînement composée exclusivement de données négatives. La situation étant très défavorable, il ne faut pas s'attendre à obtenir des performances miraculeuses (excepté dans le cas où les anomalies sont des *outliers* très remarquables du point de vue statistique), mais il est envisageable d'obtenir des méthodes opérationnelles, moyennant bien souvent un contrôle *a posteriori* par un opérateur humain.

La stratégie que nous allons utiliser ici repose sur la transformation de Karhunen-Loève, déjà étudiée dans la seconde activité (apprentissage de données fonctionnelles). Le principe est simple : on construit une base fonctionnelle orthogonale (optimale) à partir des signaux réguliers (sans anomalies) présents dans la base d'entraînement. Lorsqu'un nouveau signal est capté, on le projette dans cette base et on ne conservé que son développement mettant en jeu les p premiers coefficients, où p est classiquement choisi par compromis entre puissance expressive (mesurée par le taux de variance expliquée) et parcimonie. En général, cette dimension p est bien inférieure à la taille initiale des signaux. On reconstruit alors le signal dans sa base originale. Si cette reconstruction est de bonne qualité, c'est que le signal en question est bien adapté à la base construite avec les données d'entraînement. Dans le cas contraire, il est probable que le signal soit une anomalie.

Q1. Charger les fichiers de données `train.dat` et `test.dat` :

```
train = read.csv("datasets/tp7/train.dat")
test = read.csv("datasets/tp7/test.dat")
```

Les données étudiées sont des signaux échantillonnés en 100 points. Le jeu `train` contient 250 exemples de signaux sans anomalies. Le jeu `test` contient 270 exemples de signaux du même type, dont 20 sont des anomalies. L'objectif est de repérer ces anomalies en construisant un modèle uniquement à partir de `train`.

Notons que dans le jeu `train`, la variable `V1` vaut identiquement 0 et indique que la ligne en question n'est pas une anomalie. Dans le jeu `test`, cette variable vaut 0 pour 250 instances qui ne sont pas des anomalies, sinon, elle contient un entier positif entre 1 et 5 désignant le type de l'anomalie rencontrée.

Q2. Calculer une ACP (avec la fonction `prcomp`) sur les signaux de la base `train`. Combien de fonctions de base doit on choisir pour *capter* 99.99 % de la variance des signaux (champ `sdev` de la sortie de `prcomp`) ?

Q3. Étant donné un nouveau signal `x` issu du jeu de test :

```
x = as.numeric(test[1,2:101])
```

on peut projeter `x` sur la base définie par l'ACP avec le produit matriciel :

```
y = t(acp$rotation) %*% as.matrix(x)
```

puis tronquer la série du développement après le p-eme terme (p à fixer dans la question précédente) et opérer une reconstruction `x2` du signal original.

```
x2 = acp$rotation %*% as.matrix(y)
```

Pour vérifier que le processus de décomposition/reconstruction est correct, on pourra tracer les signaux `x` et `x2` sur une même figure :

```
plot(x, type = 'l')
lines(x2, col = 'red')
```

L'erreur de reconstructions s'évalue alors par exemple avec le RMSE de la différence entre les signaux `x` et `x2` :

```
RMSE = sqrt(mean((x-x2)^2))
```

Si cette valeur dépasse un certain seuil, on peut considérer que `x` est une anomalie.

Appliquer l'algorithme décrit ci-dessus sur les 270 données du jeu de test. Combien d'outliers parvient-on ainsi à détecter ? Combien de fausses alertes ?

Pour aller plus loin...

Corser un peu le problème en ajoutant (manuellement) de nouvelles anomalies (plus difficiles à détecter) dans le jeu de test, puis reprendre les questions précédentes.

8. Clustering et modèles à mélange... *Ou quand aucune donnée n'a d'étiquette*

Lorsque aucune donnée n'est pourvu d'étiquettes, on peut toujours essayer de partitionner (on dit alors abusivement *clusteriser* pour éviter les confusions avec le cas de l'apprentissage supervisé) les données en différents groupes.

Q1. Charger les données du fichier `cluster1.dat`, contenant les coordonnées 2D de 962 points non-étiquetés :

```
data = read.csv("datasets/tp8/cluster1.dat")
```

Q2. La méthode des *k-means*[6] propose, comme son nom le laisse deviner, de grouper les données en $k \in \mathbb{N}$ classes distinctes, de sorte à minimiser la somme des inerties de classe :

$$I = \sum_{j=1}^{k} \sum_{i=1}^{N_j} ||\mathbf{x}_{ij} - \mathbf{c}_j||^2$$

où $\mathbf{c}_j$ est le centre de gravité du groupe $j \in \{1, 2... \ k\}$, calculé à partir de ses N_j points $\mathbf{x}_{ij}$.

Le paramètre k est passé en entrée de la méthode. L'algorithme de Lloyd propose de résoudre le problème en fixant initialement k centres aléatoires, puis en itérant les deux étapes suivantes :

— chaque donnée est affectée au centre le plus proche
— on calcule les positions des nouveaux centres de gravité

Représenter les données, puis appliquer l'algorithme des *k-means* sur le jeu `data` à l'aide de la méthode (native de R) suivante :

```
model = kmeans(data, centers, algorithm = "Lloyd")
```

où le paramètre `centers` indique le nombre de centres souhaité par l'utilisateur. Le résultat du partitionnement en k groupes est contenu dans le champ `cluster` de l'objet `model`, et est référencé sous forme d'entiers compris entre 1 et k.

Q3. Représenter graphiquement le partitionnement obtenu avec des couleurs. Par exemple, pour $k = 3$ groupes :

```
cols = c("red", "forestgreen", "blue")
plot(data, col = cols[as.numeric(model$cluster)], pch = 16)
```

Q4. Recommencer l'exercice avec le jeu de données `cluster2.dat`. Comment expliquer l'échec de l'algorithme sur ce second cas d'étude ?

Q5. Une solution pour contourner le problème rencontré à la question Q4 consiste à recourir à l'algorithme Expectation-Maximization (EM), similaire à celui Lloyd, mais avec une affectation probabiliste des couleurs à chaque itération (ainsi une donnée n'est pas affectée en dur à une classe, mais partage son information statistique entre plusieurs classes, tant que l'algorithme n'a pas convergé).

Lancer le code `em.r`[7] pour observer le fonctionnement de cet algorithme.

Pour aller plus loin...

6. dite aussi méthode des moyennes mobiles, ou encore des nuées dynamiques.
7. Source : `https://gist.github.com/dirmeier/d92ea74722f6df25524a56fb7f478899`

L'algorithme EM est un grand classique de l'estimation paramétrique, et est particulièrement employé lorsque 2 quantités interdépendantes et inconnues doivent être estimées simultanément. Par exemple, dans le cadre d'un examen : connaissant les niveaux de difficulté des questions, on peut en déduire le niveau des étudiants ayant passé l'examen en analysant leurs réponses. Inversement, pour une population test d'étudiants dont le niveau est connu, l'analyse des réponses peut permettre cette fois de déterminer le niveau de difficulté des questions de l'examen. L'algorithme EM se propose de résoudre les deux problèmes simultanément par itérations successives.

Nous n'avons pas abordé ici le problème de la détermination du nombre k de classe du modèle. Il existe de nombreux critères dans la littérature (AIC, BIC, Silhouette...) qui ne posent aucune difficulté particulière d'implémentation, et nous laissons le soin au lecteur de les expérimenter si le coeur lui en dit.

9. Manifold learning... *Ou quand les données n'ont ni étiquettes, ni descripteurs*

Dans cette activité nous allons voir comment attribuer des descripteurs à un ensemble de données pour lesquelles on est seulement capable de donner des degrés de ressemblances mutuelles (c'est le cas par exemple d'un ensemble de mots clés dans un corpus de texte, que l'on ne peut que comparer mais pas situer spatialement).

On considère un ensemble de n points d'un espace métrque E : $\mathbf{x}_1,\ \mathbf{x}_2,\ ...\mathbf{x}_n$, et on note $\mathbf{M}$ la matrice des distances (de taille $n \times n$) de terme général $\mathbf{M}_{ij} = d(\mathbf{x}_i, \mathbf{x}_j)$ où $d(.,.)$ est une distance sur E.

Remarquons dans un premier temps qu'il est toujours possible d'attribuer dans n'importe quel hyperplan de $\mathbb{R}^n$ des coordonnées aux n points $(\mathbf{x}_i)_{i=1..n}$ telles que les distances consignées dans $\mathbf{M}$ soient exactement respectées. L'objectif du problème, consiste à exprimer les données $\mathbf{x}_i$ dans $\mathbb{R}^p$ où $p \in \mathbb{N}$ est un entier substantiellement inférieur à n. On appelle *réduction de dimensionnalité* cette opération. Notons qu'elle répond en pratique à deux questions distinctes :

— Exprimer des données abstraites sous formes de coordonnées dans un espace vectoriel de dimension quelconque.
— Exprimer des données de dimension n dans un espace de dimension $p << n$, en préservant au mieux les distances relatives, facilitant ainsi la représentation graphique (en particulier quand $p = 2$) et le calcul).

Nous allons utiliser ici la technique des *Laplacian Eigenmaps*, qui constitue en quelques sortes le pendant de l'ACP.

Q1. Echauffement. Lancer le script `gares.r` pour charger en mémoire une matrice `M` de taille 12×12, contenant les temps de trajet (en minutes) entre 12 gares françaises. Le vecteur `noms` contient les 12 chaînes de caractères associées.

```
source("gares.r")
```

Créer une matrice $\mathbf{D}$ de terme général :

$$\mathbf{D}_{ij} = \frac{d_{max} - \mathbf{M}_{ij}}{d_{max}}$$

où d_{max} dénote la valeur maximale de $\mathbf{M}$. Afficher $\mathbf{D}$ dans la console et vérifier que ses valeurs sont bien comprises dans $[0, 1]$, et qu'elles sont maximales sur la diagonale.

La technique des *Laplacian Eigenmaps* consiste à former la matrice :

$$\mathbf{L} = \mathbf{I}_n - \mathbf{T}^{\frac{1}{2}}\mathbf{D}\mathbf{T}^{\frac{1}{2}}$$

où $\mathbf{I}_n$ est la matrice identité de $\mathbb{R}^n$, et où $\mathbf{T}$ est une matrice diagonale de $\mathbb{R}^n$ de terme général :

$$\mathbf{T}_{ii} = \sum_{j=1}^{n} \mathbf{D}_{ij}$$

Autrement dit, $\mathbf{T}$ est une matrice diagonale de taille $n \times n$ contenant les sommes en ligne de la matrice $\mathbf{D}$.

Avec la fonction `eigen`, calculer les vecteurs propres de $\mathbf{T}$:

```
E = eigen(L)
```

Éliminer le vecteur propre correspondant à la plus petite valeur propre, puis récuperer les deux vecteurs propres suivant (dans l'ordre croissant des valeurs propres) :

```
N = nrow(M)
X = E$vectors[,c(N-2,N-1)]
```

Représenter les coordonnées de `X` dans un espace plan et vérifier que les distances entre les points dans cet espace préservent approximativement les temps de trajet entre les différentes gares.

On s'intéresse à présent à un cas d'application concret. Le fichier de données `bati.wkt` contient 50 polygones de bâtiments issus de la base de données topographique de l'IGN. L'objectif du problème consiste à *clusteriser* ces bâtiments en plusieurs groupes en procédant en 3 temps :

— Calculer de distances morphologiques entre les bâtiments. Le résultat de cet étape est consigné dans le fichier `distance.dat`.
— Appliquer la technique des *Laplacian Eigenmaps* pour représenter les bâtiments de manière abstraite dans un espace vectoriel de dimension 2 en préservant au mieux les distances calculées à l'étape précédente.
— Utiliser l'algorithme des *k-means* pour partitionner les bâtiments en k groupes (k à choisir manuellement).

Q2. Visualiser les données `bati.wkt` dans *Qgis* (menu *Couche* > *Ajouter une couche* > *Ajouter une couche de texte délimité*).

Q3. Charger les données `bati.wkt` et les distances précalculées `distances.dat` dans l'environnement R :

```
batiments = read.csv("datasets/tp9/bati.wkt", sep = ";")
distances = read.csv("datasets/tp9/distances.dat", sep = " ", h=F)
```

Q4. Appliquer la technique des *Laplacian Eigenmaps* (question Q1) pour affecter des coordonnées 2D aux bâtiments.

Q5. Avec l'algorithme des *kmeans* (TP 5, question Q2), partitionner (dans l'espace des descripteurs nouvellement créés à la question Q4) les données en k groupes.

Q6. Discuter la pertinence du partitionnement obtenu.