Classification supervisée
Méthodes d'ensemble (bagging et boosting)

Charlotte Baey (charlotte.baey@univ-lille.fr)

Protéase du VIH

On s'intéresse dans ce TP à la protéase du VIH, une enzyme qui possède un rôle majeur dans le cycle du virus du SIDA. Cette enzyme permet en effet le clivage de certaines protéines, ce qui a pour effet de libérer et de créer d'autres protéines néfastes à l'organisme et d'aider la propagation de la maladie. Pour mieux comprendre comment fonctionne la protéase du VIH, on dispose d'une base de données contenant un ensemble d'octamères (i.e. une séquence de 8 acides aminés), dont certains ont été clivés par la protéase et d'autres non.

L'objectif de l'étude est de prédire, à partir de sa séquence d'acides aminés, si un octamère est clivé ou non par la protéase du VIH.

Pré-traitement des données

Les données se trouvent dans la base `1625Data.txt`. Cette base de données contient deux variables : la première variable `Octamer` correspond à la séquence d'acides aminés composant l'octamère, et la deuxième variable `Clived` vaut 1 si l'octamère a été clivé par la protéase, et -1 sinon.

1. A partir de la séquence de 8 lettres, construire 8 variables qualitatives contenant chacune une des lettres de la séquence. **On pourra s'aider de la fonction `strsplit`**.
2. Recoder la variable cible `Clived` en *un facteur* valant 0 ou 1.
3. Quelle est la proportion d'octamères clivés ?
4. Créer une base d'apprentissage et une base de test.

Classifieur naïf

5. Construire le meilleur classifieur constant à partir de la base d'apprentissage. Calculer son taux d'erreur sur la base de test ; qui constituera un référentiel à battre avec les autres méthodes.

Arbres de décision CART

Comme les arbres CART sont à la base de la plupart des méthodes d'ensemble, on commence par étudier les performances obtenues sur un arbre individuel.

6. Construire une fonction qui prend en entrée un vecteur de prédictions `ypred` (sous la forme 0 ou 1) et un vecteur contenant les vraies classes `yobs`, et qui calcule le taux d'erreur associé.

7. Construire un arbre de décision avec la fonction `rpart`, en gardant les réglages par défaut. Calculer le taux d'erreur sur la base de test.
8. Construire un arbre de décision profond, en modifiant les options `maxdepth`, `minbucket` et `cp`.

   N.B. : On pourra stocker ces réglages dans une liste appelée `options.profond` à l'aide de la fonction `rpart.control`.

   Calculer le taux d'erreur de cet arbre.
9. Construire un arbre de décision de niveau 1 (avec seulement 1 noeud) [1].

   N.B. : Comme dans la question précédente, on pourra sauvegarder les options correspondantes dans une liste appelée `options.stump`.

   Calculer le taux d'erreur associé.
10. Commenter les résultats obtenus avec chacun des arbres. Ces résultats étaient-ils attendus ?

Bagging

Le bagging permet de réduire la variance d'un classifieur par aggrégation. Il est donc particulièrement adapté lorsque le classifieur de base a un faible biais et une forte variance. Nous allons tester (vérifier) cette propriété.

11. Charger le package `adabag`.
12. Construire un modèle de bagging à l'aide de la fonction `bagging` avec `mfinal=20` arbres, en laissant les autres réglages par défaut. Calculer le taux d'erreur associé.
13. Construire un modèle de bagging avec 20 arbres de décision à 1 noeud. On pourra réutiliser les options `options.stump` définies précédemment pour régler les paramètres des arbres du bagging (option `control=` dans la fonction `bagging`). Calculer le taux d'erreur.
14. Construire un modèle de bagging avec 20 arbres de décision profonds, et calculer le taux d'erreur associé.
15. Comparer les taux d'erreur obtenus avec chaque méthode. En sélectionnant l'une des trois approches (justifier le choix effectué), tester l'effet du nombre d'arbres, en calculant le taux d'erreur moyen sur 10 répétitions, pour des valeurs de `mfinal` égales à 1, 2, 5, 10, 20 et 50. Commenter.

Forêts aléatoires

Les forêts aléatoires sont un cas particulier d'algorithme de bagging appliqué aux arbres CART. Nous allons tester l'effet des différents réglages de l'algorithme. On a vu en cours que la pratique consistait à choisir des arbres profonds et une valeur "faible" du nombre de variables utilisées pour construire chaque arbre.

16. Charger le package `randomForest`.
17. Construire un modèle de forêt aléatoire avec `ntree=20` arbres profonds, chaque arbre étant construit avec `mtry` égal à 1, 2, 5 ou 8 variables. Tracer l'évolution du taux d'erreur moyen calculé sur 10 forêts aléatoires pour chaque valeur de `mtry`. En déduire une valeur de `mtry` optimale et commenter.

1. on parle de *decision stumps* en anglais pour désigner les arbres de décision à 1 niveau

18. Tester l'effet du nombre d'arbres en traçant l'évolution du taux d'erreur moyen calculé sur 10 forêts aléatoires, chacune construite avec la valeur optimale de `mtry` obtenue à la question précédente, en fonction de `ntree`, avec `ntree` égal à 1, 2, 5, 10, 20, 50, 100, 200 ou 500.

Boosting

On s'intéresse maintenant aux méthodes de boosting, qui contrairement aux méthodes de bagging, ne construisent pas des arbres de décision en parallèle, mais de façon séquentielle, chaque arbre se concentrant d'avantage sur les individus les moins bien classés par l'arbre précédent. Ces méthodes fonctionnent bien lorsque les classifieurs sont faibles (fort biais et donc fort variance).

AdaBoost Dans la version originelle de l'algorithme Adaboost, chaque observation est pondérée par un poids w_i, et aucune source d'aléa n'est introduite contrairement aux méthodes de bagging. Les résultats de l'algorithme sont donc déterministes : si on lance plusieurs fois la méthode sur le même jeu de données, on obtient exactement les mêmes résultats.

19. Utiliser la fonction `boosting` pour construire un modèle basé sur l'algorithme Adaboost, c'est-à-dire avec pondération des individus (option `boos`). On choisira 20 itérations de l'algorithme. Calculer le taux d'erreur.
20. Construire un nouveau modèle utilisant Adaboost, mais basé sur des arbres de décision à 1 nœud.
21. Faire de même en utilisant cette fois des arbres de décision profonds.
22. Comparer les résultats obtenus, et comparer avec la conclusion obtenue dans le cas du bagging.

XGBoost XGBoost n'est pas une méthode spécifique, mais une implémentation efficace des méthodes de *gradient boosting*. Sous `R`, cette librairie est disponible dans le package `xgboost`. Ce package fonctionne un peu différemment des précédents, et a sa propre syntaxe.

23. Charger la librairie `xgboost`.
24. Créer une table contenant les variables explicatives de l'étude sous forme de variables indicatrices ("dummy variables") correspondant à chaque catégorie. (Le faire à la main, ou utiliser par exemple la fonction `model.matrix`, en comprenant bien comment elle fonctionne !)
25. Créer deux objets à l'aide de la fonction `xgb.DMatrix`, contenant respectivement la base d'apprentissage et la base de test, et une liste contenant les deux objets, nommés respectivement `train` et `test`. Par exemple :

```
dtrainXG <- xgb.DMatrix(..., labels= ...)
dtestXG <- xgb.DMatrix(..., labels= ...)
wl <- list(train=dtrainXG, test=dtestXG)
```

26. Utiliser la fonction `xgb.train` pour appliquer un algorithme de *gradient boosting* adapté à la classification binaire, et avec `nrounds=20` itérations. Calculer le taux d'erreur.
27. Tester l'effet des paramètres de l'algorithme (`maxdepth` pour la taille des arbres, et `eta` pour le taux d'apprentissage).

Conclusion

Rédiger une conclusion générale présentant et comparant les résultats obtenus avec chaque méthode. Choisir un modèle final, et le tester sur la nouvelle base `746Data.txt`