Lec 10: Classifiers

MATH 456 - Spring 2016

Navbar: [Home] [Schedule] [Data] [Week 13 Overview] [HW Info] [Google Group]

Assigned Reading

Reading: Afifi Chapter 11. Skim the math, read the rest **carefully**. Skip/skim: 11.9-11.11

Algorithm specific information is interspersed throughout the lecture notes.

- Lots of external reading/references
- Easy to go down the rabbit hole on this stuff!
- Read/skim around to better understand these methods, but know that we could have spent easily an entire semester on machine learning algorithms.
- This is just an introduction!
- Algorithms may take a while to computer, using `cache=TRUE` in the code chunk used to fit the model is advised.

Introduction to Classifiers

We will be discussing the following 6 classifying algorithms. These are also sometimes called *Machine Learning* (ML) algorithms. They are simply methods to build a model by learning from the relationships within the data.

- Logistic Regression
- Linear Discriminant analysis
- Naieve Bayes
- Decision Trees
- Random Forests
- k-nearest neighbors

For an idea of how many algorithms are out there:

- Data Mining Map: http://www.saedsayad.com/data_mining_map.htm
- http://topepo.github.io/caret/modelList.html

Most ML algorithms will use, in some way, every variable that you give it access to. To reduce model complexity and to reduce the chance of overfitting many models apply a *penalty* to the variable that diminishes their effect on the outcome, or reduces their chance of being included into the model.

> Overfitting occurs when a statistical model describes random error or noise instead of the underlying relationship. Overfitting generally occurs when a model is excessively complex, such as having too many parameters relative to the number of observations. (Ref: https://en.wikipedia.org/wiki/Overfitting)

Machine learning in R: the `caret` package

We will be using the `caret` package in R to conduct all of our model building. Please read this small vignette for this package.

https://cran.r-project.org/web/packages/caret/vignettes/caret.pdf

Other packages you will need to install are: `pROC`, `caTools`, `bnclassify`, `mda`, `party`, `rpart.plot`, `rattle`, `penalizedLDA`.

- Typically don't need pre-install these
- You're prompted to install them when you use an algorithm that needs them
- Once installed, they will automatically load as needed.

If you want to use classifying algorithms other than the ones we have discussed here, these references will be useful:

- http://topepo.github.io/caret/bytag.html
- http://artax.karlin.mff.cuni.cz/r-help/library/caret/html/train.html

Cross-Validation

Many of these algorithms have *tuning parameters* that have to be optimized. The most common way to choose the optimal value of a tuning parameter is through *cross validation* (CV).

Recall that CV involves splitting the data (again) into testing and training data sets, running a single model over a range of values for the tuning parameter, and finding the value of the tuning parameter that provides the best fit.

We can specify how we want CV to occur in the `trainControl` function. Here we are specifying that we want to use repeated cross-validation. This means we apply k-fold CV (k defaults to 10 for this package) to one split of the data. Then repeat this process 2 more times on different rando testing/training splits of the data.

```
ctrl <- trainControl(method="repeatedcv", repeats=3,
                     classProbs = TRUE, summaryFunction = twoClassSummary)
```

The `classProbs=TRUE` argument tells the function to calculate the probabilities of being in each class (depressed or not depressed), and to provide a summary information specific to two classes (confusion matrix).

Fitting (Training) the model

The workhorse in the caret package is the `train()` function. The generic syntax looks like the following:

```
train(y ~ . ,
      data=,
      method=,
      preProc = c("center", "scale"),
      metric = ,
      trControl = ctrl)
```

- `y~.` is the model syntax
- `data` is the data set used to train the model on
- `method` is the type of machine learning algorithm used

- `preProc` is the type of pre-processing that you want to be done
- `metric` is the performance metric that you want to optimize
- `trControl` is the type of training control you want to implement.

Unless otherwise specified, this is the training control method that we will use for all algorithms explored below.

Goal: Predict Depression

Note: The data management code file for the depression data set has been updated. Go download it and update your analysis data file.

```
depress <- read.delim("C:/GitHub/MATH456/data/depress_041616.txt")
```

Pre-processing the data

Assigned Reading: http://machinelearningmastery.com/how-to-prepare-data-for-machine-learning/

Often ML algorithms perform better when the variables are also centered and scaled. This can be done at the time of calling the algorithm, so we won't do that manually here.

Recoding and ensuring variables have correct attributes

In the data management code I have already recoded variables, ensured that all factor variables were being treated appropriately. This is also the step that you would conduct a PCA or create aggregate scales like CESD (which was already created for us).

However, specifically we need to ensure that R knows `cases` is a binary class variable with positive outcome (class 1) of "Depressed". The way I handle this is to first change `cases` to a factor variable, and swap the ordering of the levels. And then I apply named values to those levels.

```
depress$cases <- factor(depress$cases, levels=c(1,0))
levels(depress$cases) <- c("Depressed", "NotDepressed")
```

Identifying missing data

```
table(is.na(depress))
```

```
##
## FALSE  TRUE
## 10876     2
```

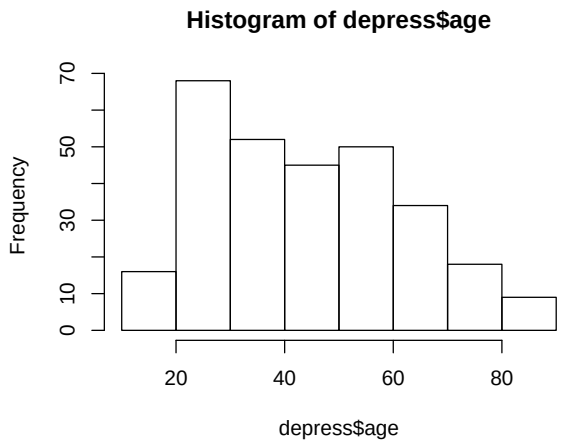
Two pieces of missing data. We have N=294, so will just delete those 2 records for now.

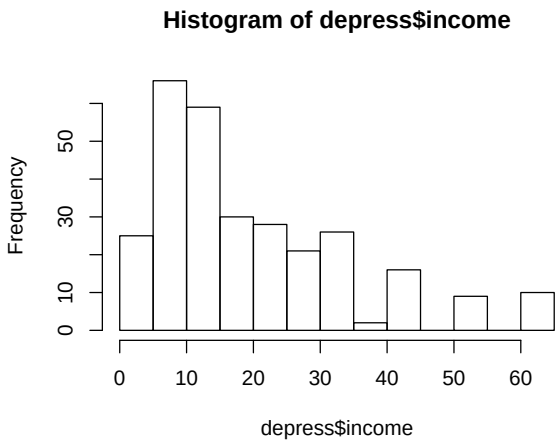
```
depress <- na.omit(depress)
```

Looking at distributions and outliers

Ensure each variable has sufficient variation.

```
par(mfrow=c(1,2))
hist(depress$age); hist(depress$income)
```





I then look at the categorical variables to see if there are any factor levels (categories) with a small number of records in them. This can lead to unstable estimates and algorithmic failure.

```
sapply(depress[,c('cases', 'marital', 'educat', 'employ', 'relig', 'health')], table)
```

```
## $cases
##
##    Depressed NotDepressed
##           49          243
##
## $marital
##
##      Divorced       Married Never Married     Separated       Widowed
##            43           126            73            12            38
##
## $educat
##
##          <HS           BS      HS Grad           MS          PhD
##           4           42          114           14            9
## Some college      Some HS
##           48           61
##
## $employ
##
##          FT Houseperson   In School       Other          PT     Retired
##         166          27           2           4          42          38
##       Unemp
##          13
```

```
##
## $relig
##
##   1   2   3   4
## 155  51  30  56
##
## $health
##
##   1   2   3   4
## 129 114  35  14
```

There are very few records with education less than HS, employed "in school" or "other". Specifically `In school` doesn't even have enough observations to calculate a variance on. I will combine "In school" with "Other".

```
library(car)
depress$employ <- recode(depress$employ, "'In School' = 'Other'")
table(depress$employ)
```

```
##
##          FT Houseperson       Other          PT     Retired       Unemp
##         166          27           6          42          38          13
```

As I was building these lecture notes, I also came across problems with the education variable (as expected). So I will further collapse categories for this variable.

```
depress$educat <- recode(depress$educat,
                         "c('<HS', 'Some HS', 'HS Grad') = 'HS';
                          c('Some college', 'BS') = 'UG';
                          c('MS', 'PhD') = 'GD'")
table(depress$educat)
```

```
##
##  GD  HS  UG
##  23 179  90
```

Similarly I will look at the average value for all the 0/1 indicator variables in the data set to get an idea of the proportion of 1's. If any have a very low percent of events (1's) in the data we will have to be mindful of how the algorithms are performing with that variable in the model.

```
sapply(depress[,c('sex', 'drink', 'regdoc', 'treat', 'beddays', 'acuteill', 'chronill')], mean)
```

```
##       sex     drink    regdoc     treat   beddays  acuteill  chronill
## 0.6198630 0.7945205 0.8150685 0.5068493 0.2157534 0.2979452 0.5068493
```

Manual variable selection.

Remove id, and the component variables `c1:c20` that are used to create `cesd`, *as well as* the `cesd` variable since `cases` is a dichotomized version of this variable.

IMPORTANT NOTE At this point I have loaded some other package that has a `select` function. If you do not specify that here you want to specifically use the select function from the `dplyr` package, this WILL NOT WORK.

```
depress <- depress %>% dplyr::select(-id, -c1:-c20, -cesd)
names(depress)
```

```
##  [1] "sex"      "age"      "marital"  "educat"   "employ"   "income"
##  [7] "relig"    "cases"    "drink"    "health"   "regdoc"   "treat"
## [13] "beddays"  "acuteill" "chronill"
```

The remaining variables are ones that I want to keep as candidate variables for a model.

Split the data into testing and training.

Instead of randomly sampling the entire data set to create the testing and training subsamples, the `createDataPartition` allows you to split the testing and training samples while stratifying on the outcome.

```
set.seed(1067)
inTrain <- createDataPartition(y=depress$cases, p=.7, list=FALSE)
train <- depress[inTrain,];dim(train)
```

```
## [1] 206  15
```

```
test  <- depress[-inTrain,];dim(test)
```

```
## [1] 86 15
```

This is advantageous in that it ensures the relative proportion of the outcome variable is the same on both the testing and training data sets.

```
prop.table(table(depress$cases))
```

```
##
##    Depressed NotDepressed
##    0.1678082    0.8321918
```

```
prop.table(table(train$cases))
```

```
##
##    Depressed NotDepressed
##    0.1699029    0.8300971
```

```
prop.table(table(test$cases))
```

```
##
##    Depressed NotDepressed
##    0.1627907    0.8372093
```

Now we are ready to build our models on the training data.

Use different classifying algorithms.

Logistic Regression

http://topepo.github.io/caret/Logistic_Regression.html

There are several different algorithms to perform a "flavor" of logistic regression analysis. We are going to use the `LogitBoost` algorithm.

Boosting: help or encourage, to increase or improve.

Boosting works on the idea that a set of *weak learners* (a classifer that is only slightly correlated with the true outcome) can be combined to create a single *strong learner* (a classifier that is strongly correlated with the true outcome).

> Boosting methods have been originally proposed as ensemble methods, ... which rely on the principle of generating multiple predictions and majority voting (averaging) among the individual classifiers
>
> Citation: https://web.stanford.edu/~hastie/Papers/buehlmann.pdf

Normal Linear Regression minimizes the error function

$$E(f) = (y(x) - f(x))^2$$

The Boosted Logistic Regression minimizes the logistic loss function

$$\sum_i log(1 + e^{-y_i f(x_i)})$$

```
LogReg <- train(cases ~ . ,
                data=train,
                method="LogitBoost",
                preProc = c("center", "scale"),
                metric = "ROC",
                trControl = ctrl)
```

Easy enough. Now let's look at the results of the model.

```
LogReg
```

```
## Boosted Logistic Regression
##
## 206 samples
##  14 predictor
##   2 classes: 'Depressed', 'NotDepressed'
##
## Pre-processing: centered (22), scaled (22)
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 186, 186, 184, 185, 186, 185, ...
## Resampling results across tuning parameters:
##
##   nIter  ROC        Sens       Spec
##   11     0.5763344  0.2222222  0.9145969
##   21     0.6110430  0.2222222  0.8909586
```

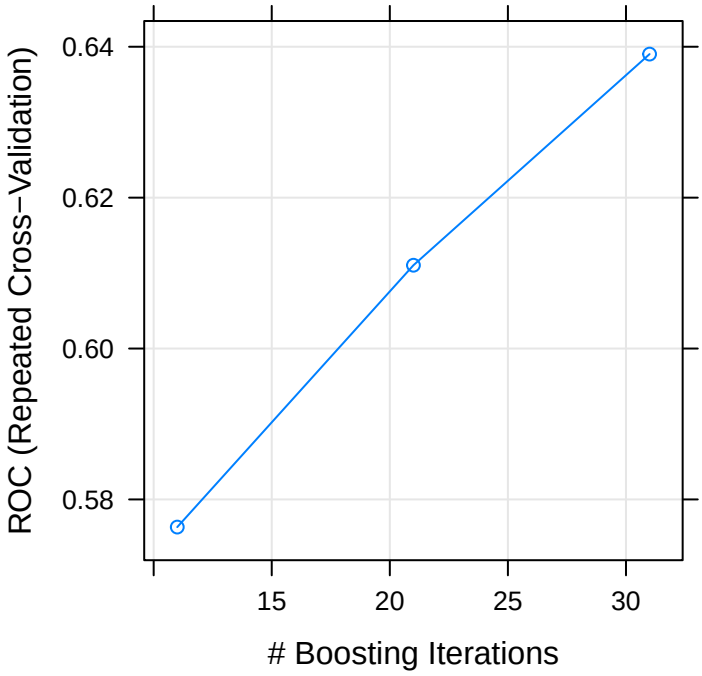
```
##   31     0.6390251  0.2916667  0.8810458
##
## ROC was used to select the optimal model using  the largest value.
## The final value used for the model was nIter = 31.
```

This output gives you a summary of the sample size going into the model, reminds you what the classes are, what pre-processing was done and what training control / optimization methods were used.

Then for each iteration the results show the sensitivity, specificity and area under the ROC curve.

We can also visualize how the AUC varied as a function of the number of boosting iterations (a tuning parameter).

```
plot(LogReg)
```



Almost all predictors available were used in the final model.

```
predictors(LogReg)
```

```
##  [1] "income"            "beddays"           "age"
##  [4] "maritalWidowed"    "employUnemp"       "employPT"
##  [7] "employHouseperson" "relig"             "sex"
## [10] "employRetired"     "educatHS"          "educatUG"
## [13] "health"
```

Additional Reading on Penalized methods http://master.bioconductor.org/help/course-materials/2003/Milan/Lectures/anestisMilan3.pdf

Discriminant analysis

- Afifi Ch 11 introduces *Linear* Discriminant analyses
- No distributional assumptions
- Finds a combination of features that separates two events
- Not all combinations are linear. http://topepo.github.io/caret/Discriminant_Analysis.html

Linear Discriminant Analysis (LDA)

- Define Z to be a linear combination of X's: $Z = a_1X_1 + a_2X_2 +, \ldots, +a_pX_p$
- Select a_i's to maximize the Mahalanobis distance between the average value of Z in group 1 from the average value of Z in group 2.

Here I use two LDA algorithms: * `LDA.1` uses the basic Linear Discriminant Analysis algorithm (`method =lda`) which does not contain any tuning parameters. * `LDA.2` includes a stepwise feature selection procedure (`method=stepLDA`)

The stepwise LDA procedure is *very* verbose (it creates a lot of output) so I specifically am NOT showing the results here. (I put `"‘{r, results='hide'}` in the code chunk header).

```
LDA.1 <- train(cases ~ . ,
               data=train,
               method="lda",
               preProc = c("center", "scale"),
               metric = "ROC",
               trControl = ctrl)
LDA.2 <- train(cases ~ . ,
               data=train,
               method="stepLDA",
               preProc = c("center", "scale"),
               metric = "ROC",
               trControl = ctrl)
```

The results show that both have high sensitivity, low specificity, with AUC values between 0.5 and 0.7.

```
LDA.1
```

```
## Linear Discriminant Analysis
##
## 206 samples
##  14 predictor
##   2 classes: 'Depressed', 'NotDepressed'
##
## Pre-processing: centered (22), scaled (22)
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 185, 185, 186, 186, 186, 185, ...
## Resampling results:
##
##   ROC        Sens       Spec
##   0.6900327  0.2083333  0.9160131
##
##
```

```
LDA.2
```

```
## Linear Discriminant Analysis with Stepwise Feature Selection
##
## 206 samples
##  14 predictor
##   2 classes: 'Depressed', 'NotDepressed'
##
## Pre-processing: centered (22), scaled (22)
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 186, 186, 185, 186, 185, 185, ...
## Resampling results:
##
##   ROC        Sens        Spec
##   0.4867102  0.01944444  0.9788671
##
## Tuning parameter 'maxvar' was held constant at a value of Inf
##
## Tuning parameter 'direction' was held constant at a value of both
##
```

The `finalModel` item within the model object contains information on the prior probabilities of being depressed and the mean value for each variable within each of the depression categories. Specifically you can also output the values for the β regression coefficients. (*Not all methods allow you to extract the values of the coefficients*)

```
print(xtable(coef(LDA.1$finalModel), digits=2), type='latex')
```

	LD1
sex	-0.12
age	0.48
maritalMarried	-0.12
maritalNever Married	-0.12
maritalSeparated	-0.17
maritalWidowed	-0.22
educatHS	-0.34
educatUG	-0.18
employHouseperson	-0.27
employOther	-0.06
employPT	-0.36
employRetired	-0.02
employUnemp	-0.36
income	0.46
relig	-0.37
drink	-0.13
health	-0.03
regdoc	0.09
treat	-0.12
beddays	-0.36
acuteill	-0.06
chronill	-0.22

- Every variable in the data set was used in this model.
- Penalized models will *shrink* these estimates down to zero for variables that do not contribute.

Compare this to the LDA with a stepwise variable selection procedure added:

```
LDA.2$finalModel
```

```
## method      : lda
## final model : y ~ sex
## <environment: 0x0000000011b6cb08>
##
## correctness rate = 0.83
```

This algorithm only chose gender as a predictor of depression.

- LDA compared to PCA: http://www.r-bloggers.com/computing-and-visualizing-lda-in-r/

Naieve Bayes

Bayesian probability models have the following form:

$$P(Y = 1|X) \propto \pi(Y = 1)P(X|Y = 1)$$

In english, this says that the posterior probability of an event occuring is proportional to the prior probability of the event occuring, times the likelihood of the event occuring given the observed data.

A naieve Bayes classifier can be thought of as fitting the joint probability model to optmize the joint likelihood $p(Y, X)$ whereas logisic regression optimizes the conditional probability $p(Y|X)$

The Naieve Bayes classifier...

- is easy and fast, performs well with multiple class (categorical) predictions.
- Assumes the effect of x_1 on P(Y=1) is independent of the value of x_2. (i.e. the X's are independent)
- Performs better than logistic regression with less training data if the assumption of conditional independence is upheld

A list of Bayesian methods included in `caret` package. * http://topepo.github.io/caret/Bayesian_Model.html

```
NB <- train(cases ~ . ,
                data=train,
                method="nb",
                preProc = c("center", "scale"),
                metric = "ROC",
                trControl = ctrl)
NB
```

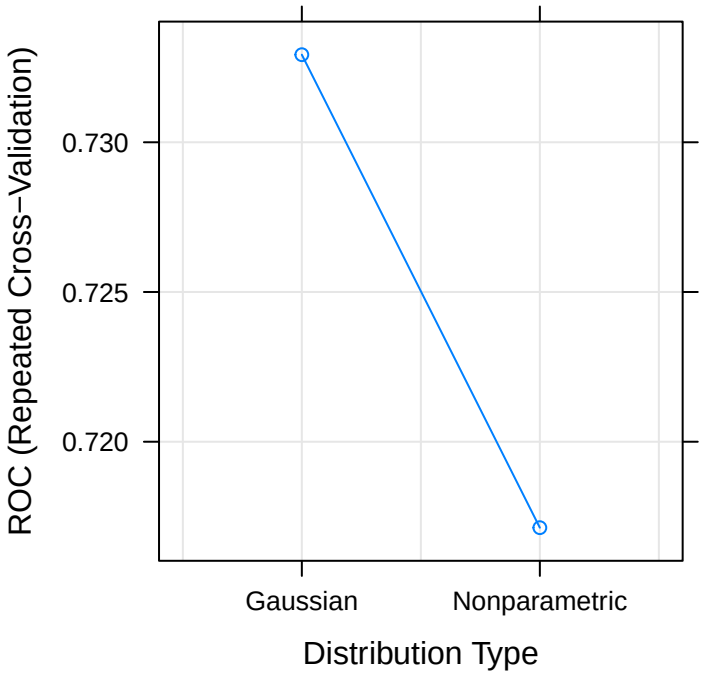
```
## Naive Bayes
##
## 206 samples
##  14 predictor
##   2 classes: 'Depressed', 'NotDepressed'
```

```
##
## Pre-processing: centered (22), scaled (22)
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 186, 185, 185, 185, 186, 185, ...
## Resampling results across tuning parameters:
##
##   usekernel  ROC        Sens       Spec
##   FALSE      0.7329238  0.3734568  0.84229
##    TRUE      0.7171296  0.0000000  1.00000
##
## Tuning parameter 'fL' was held constant at a value of 0
## Tuning
##  parameter 'adjust' was held constant at a value of 1
## ROC was used to select the optimal model using  the largest value.
## The final values used for the model were fL = 0, usekernel = FALSE
##  and adjust = 1.
```

This model takes slightly longer to fit compared to logistic regression, and at least in this example throws some warnings about zero probabilities

This plot shows you the change in AUC as a function of if the algorithm was using a Gaussian or a non-parametric kernel.

```
plot(NB)
```



There is also a slightly different set of predictors chosen in the final model.

```
predictors(NB$finalModel)
```

```
##  [1] "sex"                   "age"                   "maritalMarried"
##  [4] "maritalNever.Married" "maritalSeparated"     "maritalWidowed"
##  [7] "educatHS"              "educatUG"              "employHouseperson"
## [10] "employOther"           "employPT"              "employRetired"
## [13] "employUnemp"           "income"                "relig"
## [16] "drink"                 "health"                "regdoc"
## [19] "treat"                 "beddays"               "acuteill"
## [22] "chronill"
```

Additional References * http://www.analyticsvidhya.com/blog/2015/09/naive-bayes-explained/ * http://www.saedsayad.com/naive_bayesian.htm

Decision Trees

Tree based methods are:

- Best when the relationship between the features and the response are complex
  - lot of non-linear terms and interactions.
- Easy to explain & display graphically
- Mimics human decision making
- Tend to not have the same level of predictive accuracy as classical approaches.
- Non-parametric
- Can overfit the data more easily than other methods.

General Algorithm/Process

1. Find the variable split that best separates the categories of the response variable
2. Divide the data into two subsets based on that split
3. Within each subset, find the next variable that best separates the data into two categories.

The goal is to minimize the residual error. Technically trees can be built so tall that each observation is perfectly predicted. This is called *over fitting*. This can be avoided by *pruning* trees back (simplifying the tree).

- The level of pruning is a tuning parameter (`complexity`)
- Optimal complexity parameter can be determined by CV.

References & Readings

- https://en.wikipedia.org/wiki/Decision_tree_learning
- https://rpubs.com/ryankelly/dtrees
- http://trevorstephens.com/post/72923766261/titanic-getting-started-with-r-part-3-decision

The last article from Trevor Stephens does a *very* good job explaining and interpreting Decision Trees using the Titanic data.

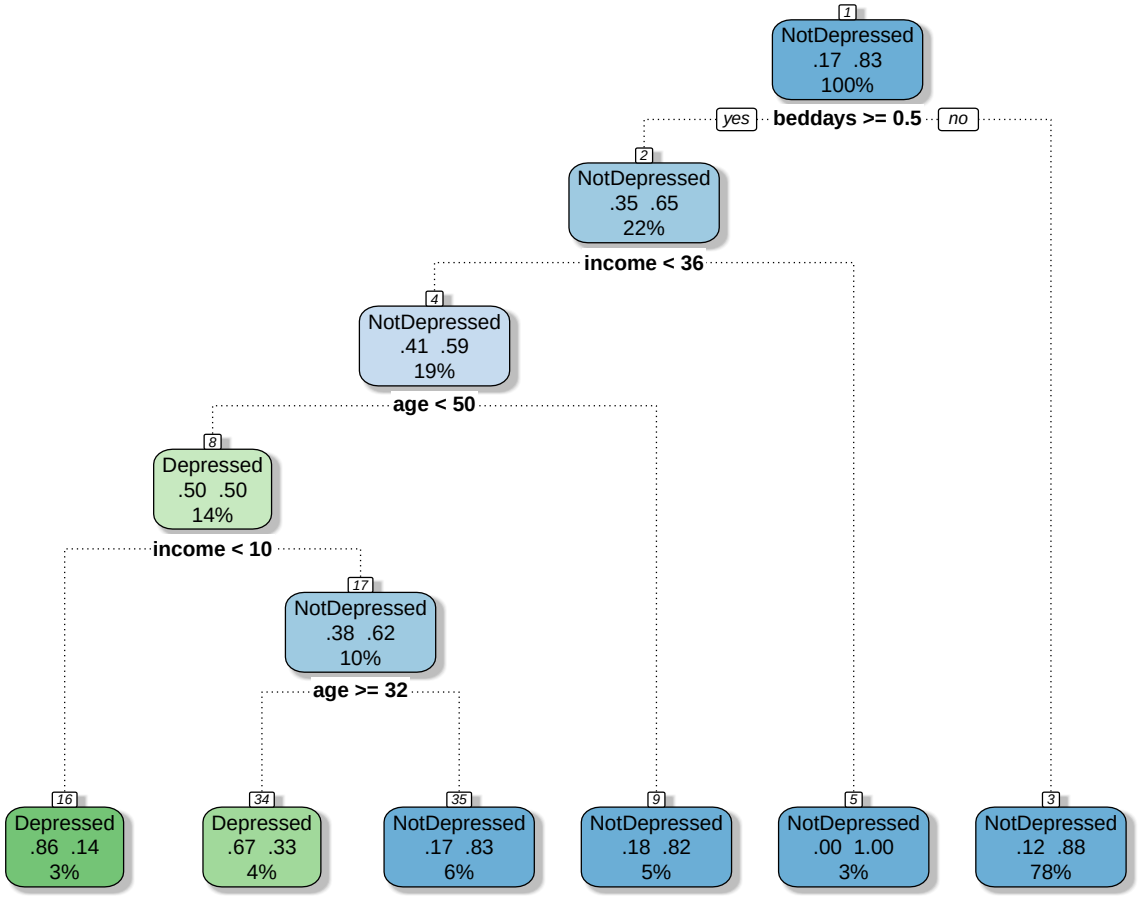
```
DT <- train(cases ~ . ,
                data=train,
                method="rpart",
                metric = "ROC",
                trControl = ctrl)
DT
```

```
## CART
##
## 206 samples
##  14 predictor
##   2 classes: 'Depressed', 'NotDepressed'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 186, 185, 186, 185, 185, 185, ...
## Resampling results across tuning parameters:
##
##   cp          ROC        Sens        Spec
##   0.00000000  0.6030773  0.10277778  0.8863834
##   0.01785714  0.5811819  0.07222222  0.8981481
##   0.03571429  0.5440087  0.05555556  0.9334423
##
## ROC was used to select the optimal model using  the largest value.
## The final value used for the model was cp = 0.
```

Visualize the decision tree.

Note: Mac users may have to install X11 as part of installing `rattle`. If you run into problems, either come see me or don't use the `rattle` library. Then you'll have to replace `fancyRpartPlot` with a simple `plot()`. Another common error during installation that may occur will give you the following error: "The program can't start because libatk-1.0.0.dll is missing from your computer. Try reinstalling the program to fix the problem. Hit"OK" and install gTk2. You may have to restart R.

```
library(rpart.plot); library(rattle)
fancyRpartPlot(DT$finalModel, sub="")
```



The boxes show several pieces of information:

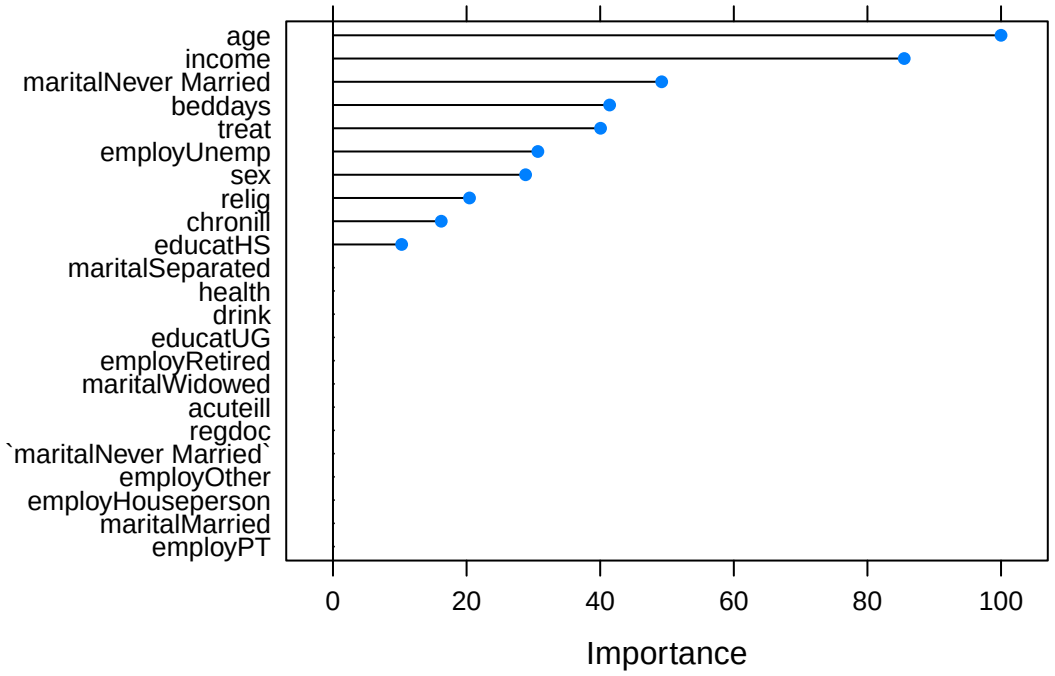
- The name at the top is the predicted class for all observations in that node.
- There are two colors to the plot: Blue for one group and Green for the other. The level of color intensity is a measure of the node "purity". A pure node is one that contains all observations of one class
- The percentages in the third row show the proportion of observations in each class.
- The bottom percentage is how many observations are in this node.

Below each node is the splitting criteria. This is where having proper variable names and factor levels makes this plot easier to read.

This tree is grown to a certain length, one that balances node purity and model complexity. The larger/deeper/more complex the tree, the higher chance you will overfit the data.

Tree based models also provide a measure on the importance of each candidate variable. These can be visualized using a **Variable Importance Plot**.

```
plot(varImp(DT))
```



This echo's the decision tree plot in that top three primary variables important in predicting depression is age, beddays, chronill.

Random Forests

- Grow a lot of decision trees, each more complex/deeper than above.
- Add a randomness component to ensure each tree grown is different.
  - Randomly sample with replacement a certiain % of observations (Bootstrapping)
  - Randomly sample a smaller group (usually $\sqrt{(p)}$) of candidate variables.
- Average the outcomes across groups. Also called voting majority or Aggregation

FYI: Bootstrapping + Aggregation == Bagging.

A bonus feature of bagging is that the sample of records that were not chosen to build the tree on (called the Out of Bag (OOB) sample) acts as a testing data sample to assess how well each tree is performing.

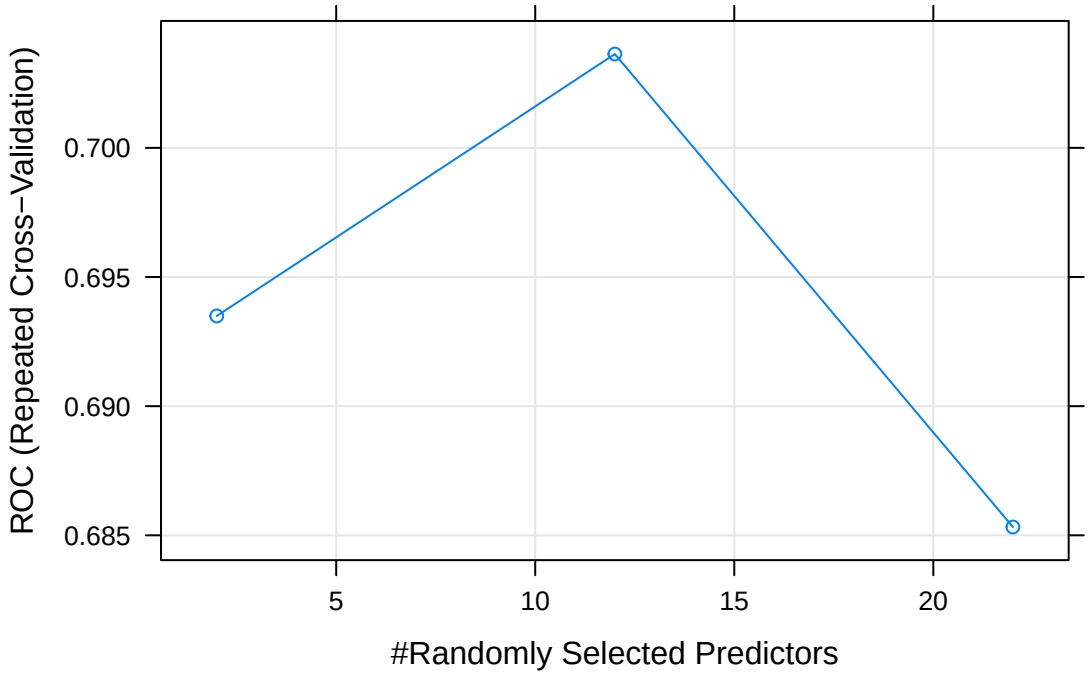
Additional Reading http://trevorstephens.com/post/73770963794/titanic-getting-started-with-r-part-5-random

```
RF <- train(cases ~ . ,
                data=train,
                method="rf",
                metric = "ROC",
                trControl = ctrl)
RF
```

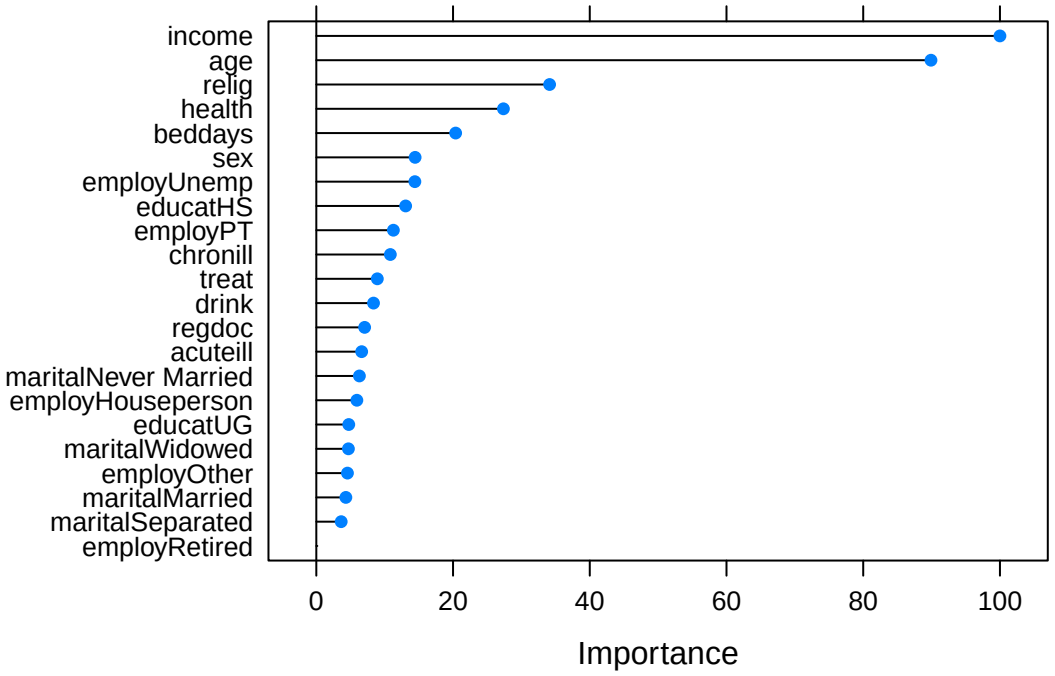
```
## Random Forest
##
```

```
## 206 samples
##  14 predictor
##   2 classes: 'Depressed', 'NotDepressed'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 184, 186, 185, 185, 186, 186, ...
## Resampling results across tuning parameters:
##
##   mtry  ROC        Sens        Spec
##    2    0.6934913  0.01944444  1.0000000
##   12    0.7036311  0.15000000  0.9746187
##   22    0.6853214  0.14166667  0.9629630
##
## ROC was used to select the optimal model using  the largest value.
## The final value used for the model was mtry = 12.
```

```
plot(RF)
```




```
plot(varImp(RF))
```



k-nearest neighbors

- Calculate the euclidean distance between a point with unknown class and every other point with known class membership.
- Using a majority vote, assign the class label to the unknown point based on a majority vote of the k nearest neighbors.

PROS * Unbiased * Easy to understand * Non-parametric

CONS * Too simple

Additional Reading

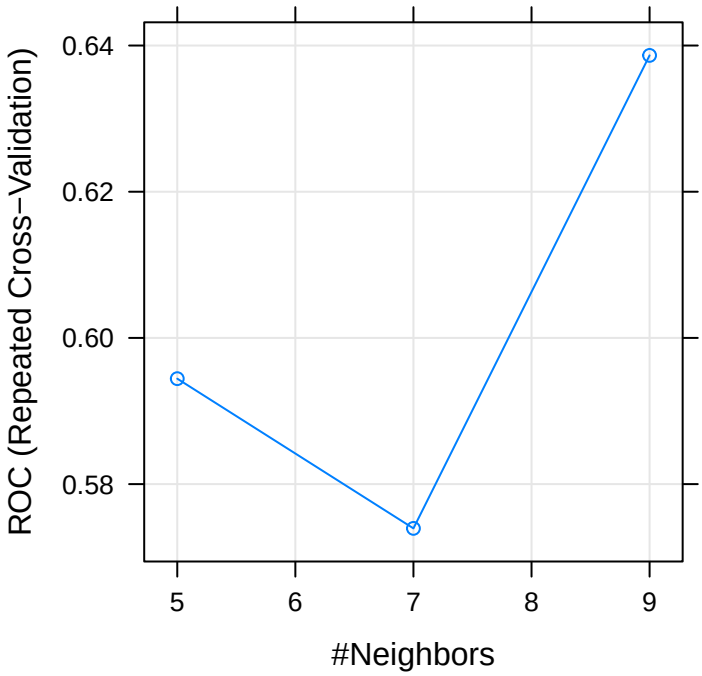
- https://www.datacamp.com/community/tutorials/machine-learning-in-r
- http://www.analyticsvidhya.com/blog/2015/08/learning-concept-knn-algorithms-programming/

```
kNN <- train(cases ~ . ,
                data=train,
                method="knn",
                metric = "ROC",
                trControl = ctrl)
kNN
```

```
## k-Nearest Neighbors
##
## 206 samples
##  14 predictor
##   2 classes: 'Depressed', 'NotDepressed'
```

```
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 185, 184, 186, 186, 186, 186, ...
## Resampling results across tuning parameters:
##
##   k  ROC        Sens        Spec
##   5  0.5944263  0.09166667  0.9747277
##   7  0.5739424  0.08333333  0.9862745
##   9  0.6386483  0.05555556  0.9882353
##
## ROC was used to select the optimal model using  the largest value.
## The final value used for the model was k = 9.
```

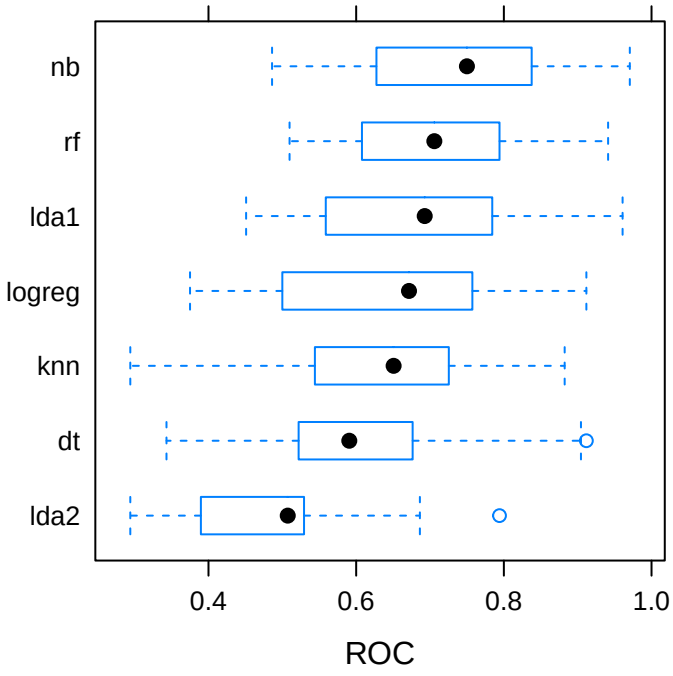
```
plot(kNN)
```



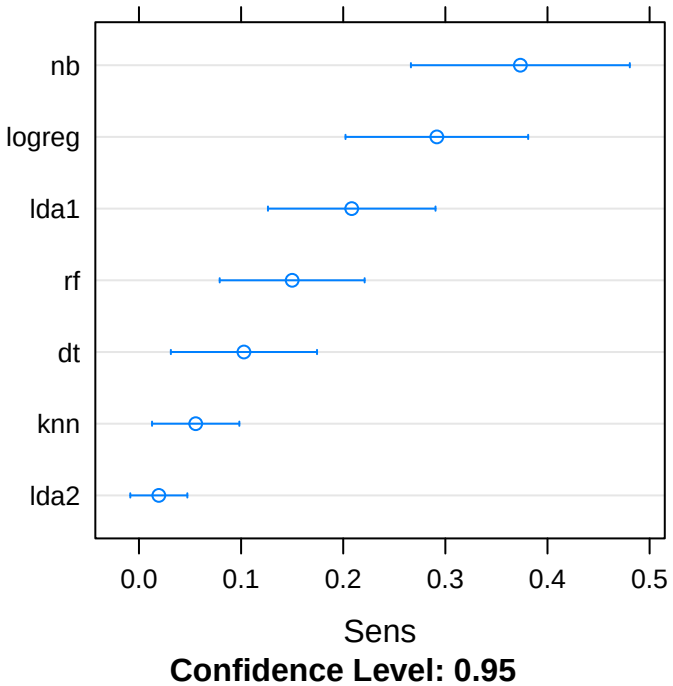
Compare algorithm performance

A resampling technique can be used to compare model performance. (Ref: _Hothorn at al, "The design and analysis of benchmark experiments- Journal of Computational and Graphical Statistics (2005) vol 14 (3) pp 675-699)_)

```
rValues <- resamples(list(logreg=LogReg,lda1=LDA.1, lda2=LDA.2, nb=NB, dt=DT, rf=RF, knn=kNN))
bwplot(rValues,metric="ROC")
```




```
dotplot(rValues,metric="Sens")
```



Confidence Level: 0.95

```
# Hint: Look at rValues$metrics for other metrics you can visualize & compare.
```

We can compare algorithms on their accuracy predicting the training data set. Here I create a confusion matrix by using each model to predict the class membership of data on the training data set.

```
acc.log <- confusionMatrix(predict(LogReg, train), train$cases, positive='Depressed')
acc.ld1 <- confusionMatrix(predict(LDA.1, train), train$cases, positive='Depressed')
acc.ld2 <- confusionMatrix(predict(LDA.2, train), train$cases, positive='Depressed')
acc.nb  <- confusionMatrix(predict(NB, train), train$cases, positive='Depressed')
acc.dt  <- confusionMatrix(predict(DT, train), train$cases, positive='Depressed')
acc.rf  <- confusionMatrix(predict(RF, train), train$cases, positive='Depressed')
acc.knn <- confusionMatrix(predict(kNN, train), train$cases, positive='Depressed')
```

Ok so how do extract the accuracy values? Lets see what data is in the object created by the `confusionMatrix` function.

```
names(acc.log)
```

```
## [1] "positive" "table"    "overall"  "byClass"  "dots"
```

```
names(acc.log$overall)
```

```
## [1] "Accuracy"       "Kappa"          "AccuracyLower"  "AccuracyUpper"
## [5] "AccuracyNull"   "AccuracyPValue" "McnemarPValue"
```

```
names(acc.log$byClass)
```

```
## [1] "Sensitivity"          "Specificity"          "Pos Pred Value"
## [4] "Neg Pred Value"       "Prevalence"           "Detection Rate"
## [7] "Detection Prevalence" "Balanced Accuracy"
```

Now that I've identified that the value for Accuracy is the first element in the `$overall` list, I can extract these values and compare them in a tabular format. Furthermore the sensitivity, specificity and other table values can be found in the `$byClass` list.

For berivity sake (and to demonstrate some advanced R programming techniques) I create a function `getmetrics` to extract only the values of the performance metrics I am interested in, then apply that function to all models in a list.

```
library(foreach)
model.list <- list(acc.log, acc.ld1, acc.ld2, acc.nb, acc.dt, acc.rf, acc.knn)
getmetrics <- function(x){
  y <- x[[1]]
  metrics <- c(y$overall[1], y$byClass[1:2])
  return(metrics)
  }

Metrics <- foreach(i = 1:length(model.list), .combine=rbind) %do% getmetrics(model.list[i])
rownames(Metrics) <- c("Boosted Logistic Regression", "LDA", "LDA w/stepwise",
               "Naieve Bayes", "Decision Tree", "Random Forest", "k-Nearest neighbors")
```

	Accuracy	Sensitivity	Specificity
Boosted Logistic Regression	0.883	0.486	0.965
LDA	0.874	0.429	0.965
LDA w/stepwise	0.830	0.000	1.000
Naieve Bayes	0.791	0.429	0.865
Decision Tree	0.869	0.343	0.977
Random Forest	1.000	1.000	1.000
k-Nearest neighbors	0.845	0.086	1.000

Making predictions

On the hold-out testing data set.

```
confusionMatrix(predict(RF, test), test$cases, positive='Depressed')
```

```
## Confusion Matrix and Statistics
##
##               Reference
## Prediction     Depressed NotDepressed
##   Depressed            3            4
##   NotDepressed        11           68
##
##                Accuracy : 0.8256
##                  95% CI : (0.7287, 0.899)
##     No Information Rate : 0.8372
##     P-Value [Acc > NIR] : 0.6788
##
##                   Kappa : 0.1988
##  Mcnemar's Test P-Value : 0.1213
##
##             Sensitivity : 0.21429
##             Specificity : 0.94444
##          Pos Pred Value : 0.42857
##          Neg Pred Value : 0.86076
##              Prevalence : 0.16279
##          Detection Rate : 0.03488
##    Detection Prevalence : 0.08140
##       Balanced Accuracy : 0.57937
##
##        'Positive' Class : Depressed
##
```

The Random Forest model had an accuracy of 0.826(0.729 ,0.899) on the testing data set.

Additional references

- http://www.statmethods.net/advstats/discriminant.html
- http://rstudio-pubs-static.s3.amazonaws.com/35817_2552e05f1d4e4db8ba87b334101a43da.html
- https://www.youtube.com/watch?v=s8pvp2Ctxfc
- http://www.r-bloggers.com/in-depth-introduction-to-machine-learning-in-15-hours-of-expert-videos/

http://michael.hahsler.net/SMU/EMIS7332/R/chap5.html

On Your Own

Using the Parental HIV data set, build a predictive model for whether or not a student skips class `HOOKEY` using all 6 of the classifying algorithms discussed in this set of lecture notes.