Practical machine learning - Notes

Tanner Prestegard

Course taken from 9/7/2015 - 10/4/2015

Motivation and prerequisites

- Basic ideas behind machine learning/prediction
  - Study design: training vs. test sets.
  - Conceptual issues: out of sample error, ROC curves.
  - Practical implementation: the caret package.
- Who predicts things?
  - Governments: pension payments.
  - Google: whether you will click on an ad.
  - Amazon: what movies you will watch.
  - Insurance companies: what your risk of death is.
  - Johns Hopkins: who will succeed in their programs.

What is prediction?

- Components of a predictor:
  - Question.
  - Input data.
  - Features.
  - Algorithm.
  - Parameters.
  - Evaluation.

Relative order of importance

- Defining the question is the most important step!
- Input data: garbage in = garbage out.
  - May be easy: movie ratings -> new movie ratings.
  - May be hard: gene expression data -> disease.
  - Depends on how you define a "good prediction."
  - Often more data helps more than better models.
  - Very important to collect the "right" data that is relevant to your question.

- Features matter!
  - Properties of good features:
    - Lead to data compression.
    - Retain relevant information.
    - Are created based on expert application knowledge.
  - Common mistakes:
    - Trying to automate feature selection.
    - Not paying attention to data-specific quirks.
    - Throwing away information unnecessarily.
- Algorithms matter less than you'd think.
- Issues to consider: your method should be interpretable, simple, accurate, fast (to train and test), and scalable.
- Prediction is about accuracy tradeoffs:
  - Interpretability versus accuracy.
  - Speed versus accuracy.
  - Simplicity versus accuracy.
  - Scalability versus accuracy.

In sample and out of sample errors

- In sample error: the error rate you get on the same data set that you used to build your predictor. Sometimes called resubstitution error. Usually slightly optimistic.
- Out of sample error: the error rate you get on a new data set. Sometimes calles generalization error.
- Key ideas:
  - Out of sample error is what you really care about.
  - In sample error $<$ out of sample error, due to overfitting (matching your algorithm to the data you have).
- Data have two parts: signal and noise.
  - The goal of a predictor is to find signal.
  - You can always design a perfect in-sample predictor, but you capture both signal and noise when you do that.
  - This predictor won't perform as well on new samples (overfitting again).

Prediction study design

- Define your error rate.
- Split data into: training, testing, and validation (optional) datasets.
- On the training set, pick features and use cross-validation.
- On the training set, pick a prediction function and use cross-validation.
- If no validation, apply the function once to the test set.

- If using validation, apply the function to the test set and refine, then apply once to the validation dataset.
- Avoid small sample sizes
  - Example: predicting a binary outcome, like flipping a coin.
  - Probability of perfect classification is approximately $(1/2)^{\text{sample size}}$:
    - $n = 1$: flipping a coin gives 50% chance of 100% accuracy.
    - $n = 2$: flipping a coin gives 25% chance of 100% accuracy.
    - $n = 100$: flipping a coin gives 0.1% chance of 100% accuracy.
- Rules of thumb for prediction study design
  - If you have a large sample size:
    - 60% training.
    - 20% test.
    - 20% validation.
  - If you have a medium sample size:
    - 60% training.
    - 40% testing.
  - If you have a small sample size:
    - Do cross-validation.
    - Report caveats of small sample size.
- Some principles to remember:
  - Set the test/validation set aside and don't look at it!
  - In general, randomly sample the training and test datasets.
  - Your datasets must reflect the structure of the problem: if predictions evolve with time, split the training/test by time chunks (called backtesting in finance).
  - All subsets should reflect as much diversity as possible.
    - Random assignment does this.
    - You can also try to balance by features, but this is tricky.

Types of errors

- Positive = identified, negative = rejected.
  - True positive (TP): correctly identified signal.
  - False positive (FP): incorrectly identified noise as signal.
  - True negative (TN): correctly rejected noise.
  - False negative (FN): incorrectly rejected signal as noise.
  - Sensitivity: Pr(positive test | sick person) = TP / (TP+FN)
  - Specificity: Pr(negative test | healthy person) = TN / (FP + TN)
  - Positive predictive value: Pr(sick person | positive test) = TP / (TP + FP)
  - Negative predictive value: Pr(healthy person | negative test) = TN / (FN + TN)
  - Accuracy: Pr(correct outcome) = (TP + TN) / (TP + FP + FN + TN)
- For continuous data, there are a few ways to handle this.

  - Mean squared error (MSE): $MSE = \frac{1}{n}\sum_{i=1}^{n}\left(Prediction_i - Truth_i\right)^2$ or root mean square error (RMSE): $RMSE = \sqrt{MSE}$.
    - Continuous data, sensitive to outliers (outliers may raise the mean significantly).
  - Median absolute deviation.
    - Continuous data, often more robust.
  - Sensitivity: if you want few positives called negatives.
  - Specificity: if you want few negatives called positives.
  - Accuracy: weights false positives and negatives equally.
  - Concordance.

Receiver operating characteristic (ROC) curves

- Why a curve?
  - In binary classification you are predicting one of two categories.
  - But your predictions are ofen quantitative: probability of this or that.
  - The *cutoff* you choose gives different results.
- ROC curves:
  - X-axis: 1 - specificity, or probability of being a false positive.
  - Y-axis: probability of being a true positive.
  - To compare different curves, you can calculate the total area under each curve (more area generally means a better predictor).
    - Area under curve = 0.5 is equivalent to random guessing.
    - Area under curve = 1 is a perfect classifier.
    - In general, if your area under the curve is more than 0.8, that is considered "good."

Cross-validation

- Key ideas:
  - Accuracy on the training set (resubstitution accuracy) is optimistic.
  - A better estimate comes from an independent dataset (test set accuracy).
  - But we can't use the test set when building the model or it becomes part of the training set.
  - So we estimate the test set accuracy with the training set.
- Cross-validation approach:
  - Use the training set.
  - Split it into training/test sets.
    - Use random subsampling to do this.
    - Can also do "K-fold" cross-validation.
    - Another option: "leave one out". Use only one sample for test dataset and the rest for training; repeat with all samples.
  - Build a model on the training set.
  - Evaluate on the test set.
  - Repeat and average the estimated errors.

- Useful for:
  - Picking variables to include in the model.
  - Picking the type of prediction function to use.
  - Picking the parameters in the prediction function.
  - Comparing different predictors.
- Considerations:
  - For time-series data, you must use chunks of data.
  - For K-fold cross-validation.
    - Larger K: less bias, more variance.
    - Smaller K: more bias, less variance.
  - Random sampling must be done without replacement.
  - Random sampling with replacement is called *bootstrapping.*
    - Underestimates the error.
    - Can be corrected, but it's complicated (see 0.632 Bootstrap rule).
  - If you cross-validate to pick predictors, you must estimate errors on independent data.

What data should you use?

- Key idea: to predict X, use data as closely related to X as you possibly can. (example: Moneyball; use player performance data to predict player performance)
- Using unrelated data is the most common mistake!

The caret package

- Short for "Classification And REgression Training)
- Streamlines the process for creating predictive models.
- Functionality
  - Some pre-processing/cleaning: `preProcess`
  - Data splitting: `createDataPartition, createResample, createTimeSlices`
  - Training/testing functions: `train, predict`
  - Model comparison: `confusionMatrix`
- Machine learning algorithms in R
  - Linear discriminant analysis
  - Regression
  - Naive Bayes
  - Support vector machines
  - Classification and regression trees
  - Random forests
  - Boosting
  - Etc.

- Example:

  ```
  library(caret); library(kernlab); data(spam)
  ## Divide data into training and test sets.
  ## Split on data type, 75% training, 25% testing.
  inTrain <- createDataPartition(y=spam$type, p=0.75, list=FALSE)
  training <- spam[inTrain,]
  testing <- spam[-inTrain,]
  ## Fit a model.
  set.seed(32343)
  modelFit <- train(type ~., data=training, method="glm")
  ## Look at final model.
  modelFit$finalModel
  ## Test on new samples.
  predictions <- predict(modelFit, newdata=testing)
  ## Example: confusion matrix. Useful for getting several accuracy measures.
  confusionMatrix(predictions, testing$type)
  ```

Data slicing

- Example: K-fold

  ```
  set.seed(32323)
  folds <- createFolds(y=spam$type, k=10, list=TRUE, returnTrain=TRUE)
  sapply(folds, length)
  ## returnTrain=TRUE returns the training and testing sets,
  ## returnTRAIN=FALSE returns only the testing set.
  ```

- Example: resampling (with replacement)

  ```
  set.seed(32323)
  folds <- createResample(y=spam$type, times=10, list=TRUE)
  sapply(folds, length)
  ```

- Example: time slices

  ```
  set.seed(32323)
  tme <- 1:1000
  folds <- createTimeSlices(y=tme, initialWindow=20, horizon=10)
  ```

Training options

- Use `args(train.default)` to see all available options.
- Use `args(trainControl)` to see other options for training setup.
- Continuous metric options:
  - RMSE: root mean squared error.
  - RSquared: R^2 from regression models.
- Categorical outcomes:
  - Accuracy: fraction correct.
  - Kappa: a measure of concordance.

- `trainControl` resampling
  - Method:
    - boot: bootstrapping.
    - boot632: bootstrapping with adjustment.
    - cv: cross-validation.
    - repeatedcv: repeated cross-validation.
    - LOOCV: leave one out cross-validation.
  - Number:
    - For boot/cross-validation.
    - Number of subsamples to take.
  - Repeats:
    - Number of times to repeate subsampling.
    - If big, this can slow things down.
- Setting the seed:
  - It is often useful to set an overall seed.
  - You can also set a seed for each resample.
  - Seeding each resample is useful for parallel fits.

Plotting predictors

- For this example, we will use the wages data (ISLR package).
- Feature plot (from caret package): `featurePlot(x=training[,c(“age”,”education”,”jobclass”)], y=training$wage, plot=”pairs”)`
- Can use ggplot to plot by category (using color), plotting regression smoothers, etc.
- Can use `cut2` (from Hmisc package) to make factors:
  - `cutWage <- cut2(training$wage, g=3) ## g=3 implies 3 groups`
- Another useful plot: box plots with points overlaid.
- Tables are useful.
  - `prop.table()` gives the proportion in each category.
- Density plots are useful for continuous predictors.
  - Example: `qplot(wage, color=education, data=training, geom=”density”)`
- Notes:
  - Make your plots only with the training set!
  - Things you should be looking for:
    - Imbalance in outcomes/predictors.
    - Outliers.
    - Groups of points not explained by any of the predictors.
    - Skewed variables.

Pre-processing

- Why preprocess?
  - Sometimes certain predictors may have high variances, weird skews, etc.
  - These features can cause problems for machine learning algorithms.
  - You want to simplify things; the algorithm will likely produce better results this way.
- Standardizing: `(x - mean(x))/sd(x)`
  - Produces variables with mean 0 and standard deviation 1.
  - If we do this in the training, we have to do it to the test set, using the mean and SD of the *training* set!
  - `preProcess` function: `preObj <- preProcess(training[,-58], method=c("center","scale"))`
- Can pass the `preProcess` command directly to the `train()` function:
  - `modelFit <- train(type ~., data=training, preProcess=c("center","scale"), method="glm")`
- Box-Cox transforms: a set of transformations which take continuous data and try to make them look like normal data.
  - Uses MLE methods.
  - Can be used with `preProcess()`.
- Imputing data: prediction algorithms will likely fail if there is missing data.
  - `preObj <- preProcess(training[,-58], method="knnImpute")`
- Remember: the training and test datasets must be processed in the same way!
- Also: be careful when transforming factor variables!

Covariate creation

- Covariates: also known as predictors or features.
- Two levels of covariate creation:
  - Level 1: from raw data to covariate.
    - Depends heavily on the application.
    - The balancing act is summarization vs. information loss.
    - Examples:
      - Text files: frequency of words, phrases, capital letters, etc.
      - Images: edges, corners, blobs, ridges.
      - Webpages: number and type of images, position of elements, colors, videos.
      - People: height, weight, hair color, gender, country of origin.
    - The more knowledge of the system you have, the better job you will do.
    - When in doubt, err on the side of more features.
    - Can be automated, but use caution!
  - Level 2: transforming tidy covariates.
    - More necessary for some methods (regression, svms, etc.) than for others (classification trees).
    - Should be done only on the training set!

    - The best approach is through exploratory analysis (plotting/tables).
    - New covariates should be added to data frames.

- Common covariates to add/dummy variables:
  - Basic idea: convert factor variables to indicator variables. (quantitative information easier for algorithms to use than qualitative)

    ```
    dummies <- dummyVar(wage ~ jobclass, data=training)
    head(predict(dummies, newdata=training)
    ```

  - Removing zero covariates: features with no variability (same for all cases) are not useful.

    ```
    nsv <- nearZeroVar(training, saveMetrics=TRUE)
    ```

- Spline basis - instead of fitting a linear prediction function, you can use a "curvy" line.

  ```
  library(splines)
  bsBasis <- bs(trainingage, df=3) ## degree 3 polynomial
  # Add model
  lm1 <- lm(wage ~ bsBasis, data=training)
  plot(training$age, training$wage, pch=19, cex=0.5)
  points(training$age, predict(lm1, newdata=training), col="red", pch=19, cex=0.5)
  ## Apply to the test dataset
  predict(bsBasis, age=testing$age)
  ```

- Notes and further reading
  - Level 1 feature creation:
    - Science is key. Google "feature extraction for [data type]."
    - Err on overcreation of features.
    - In some applications (images, voices), automated feature creation is possible/necessary.
  - Level 2 feature creation:
    - The function `preProcess` in caret will handle some preprocessing.
    - Create new covariates if you think they will improve the fit.
    - Use exploratory analysis on the training set for creating them.
    - Be careful about overfitting.
  - If you want to fit spline models, use the `gam` method in the caret package, which allows smoothing of multiple variables.

Preprocessing with Principal Components Analysis (PCA)

- Useful when many predictors are correlated.

-
  ```
  library(caret); library(kernlab); data(spam)
  inTrain <- createDataPartition(y=spam$type, p=0.75, list=FALSE)
  training <- spam[inTrain,]
  testing <- spam[-inTrain,]
  ## Calculate correlation between variables.
  M <- abs(cor(training[,-58]))
  diag(M) <- 0 ## Ignore self-correlations.
  which(M > 0.8, arr.ind=TRUE)
  ```

- Basic PCA idea:

  - We might not need every predictor.
  - A weighted combination of correlated predictors might be better.
  - We should pick this combination to capture the "most information" possible.
  - Benefits: reduced number of predictors, reduced noise (due to averaging).

- How to combine?
  - You could rotate (i.e., add / subtract two variables).

- Related problems:
  - You have multivariate variables $X_1, ..., X_n$ such that $X_1 = (X_{11}, ..., X_{1m})$.
  - Find a new set of multivariate variables that are uncorrelated and explain as much variance as possible.
  - If you put all the variables together in one matrix, find the best matrix created with fewer variables (lower rank) that explains the original ndata.
  - The first goal is statistical and the second goal is data compression.

- Related solutions:
  - Singular value decomposition (SVD):
    * If X is a matrix with each variable in a column and each observation in a row, then the SVD is a matrix decomposition: $X = UDV^T$, where the columns of U are orthogonal (left singular vectors), the columns of V are orthogonal (right singular vectors), and D is a diagonal matrix (singular values).
  - Principal components analysis (PCA):
    * The principal components are equal to the right singular values if you first scale (subtract the mean, divide by the standard deviation) the variables.
    * Example:

```
smallSpam <- spam[,c(34,32)]
prComp <- prcomp(smallSpam)
plot(prComp$x[,1],prComp$x[,2])
## Look at rotation matrix
prComp$rotation
```

    * PCA with caret package:

```
preProc <- preProcess(log10(spam[,-58]+1), method="pca", pcaComp=2)
trainPC <- predict(preProc, log10(spam[,-58]+1))
modelFit <- train(training$type ~ ., method="glm", data=trainPC)
```

- Most useful for linear-type models.
- Can make it harder to interpret predictors.
- Watch for outliers!
  - Transform first (with logs/Box-Cox).
  - Plot predictors to identify problems.

Predicting with regression

- Key ideas:
  - Fit a simple linear regression model.
  - Plug in new covariates and multiply by the coefficients.
  - Useful when the linear model is (nearly) correct.
- Pros: easy to implement and interpret.
- Cons: often poor performance in nonlinear settings.
- Example: Old Faithful eruptions

```
library(caret); data(faithful); set.seed(333)
inTrain <- createDataPartition(y=faithful$waiting, p=0.5, list=FALSE)
trainFaith <- faithful[inTrain,]; testFaith <- faithful[-inTrain,]
## Fit a linear model
lm1 <- lm(eruption ~ waiting, data=trainFaith
## plot
plot(trainFaith$waiting, trainFaith$eruptions, pch=19, col="blue", xlab="Waiting", yla
lines(trainFaith$waiting, lm1$fitted, lwd=3)
## Predict a new value
coef(lm1)[1] + coef(lm1)[2]*80
## Can do this as well
newdata <- data.frame(waiting = 80)
predict(lm1,newdata)
## Plot predictions - training and test
par(mfrow=c(1,2))
plot(trainFaith$waiting,trainFaith$eruptions,pch=19,col="blue",xlab="Waiting",ylab="Du
lines(trainFaith$waiting,predict(lm1),lwd=3)
plot(testFaith$waiting,testFaith$eruptions,pch=19,col="blue",xlab="Waiting",ylab="Dura
lines(testFaith$waiting,predict(lm1,newdata=testFaith),lwd=3)
## Get training set/test set errors
## RMSE
sqrt(sum((lm1$fitted-trainFaith$eruptions)^2))
sqrt(sum((predict(lm1,newdata=testFaith)-testFaith$eruptions)^2))
## Prediction intervals
pred1 <- predict(lm1,newdata=testFaith, interval="prediction")
ord <- order(testFaith$waiting)
plot(testFaith$waiting,testFaith$eruptions,pch=19,col="blue")
matlines(testFaith$waiting[ord],pred1[ord,],type="l",col=c(1,2,2),lty=c(1,1,1),lwd=3)
## Same process with caret
modFit <- train(eruptions ~ waiting, data=trainFaith, method="lm")
summary(modFit$finalModel)
```

- Regression models with multiple covariates can be included.
- Often useful in combination with other models.

Predicting with regression - multiple covariates

- Important to decide which predictors are most useful
- Example: wage data

```
library(ISLR); library(ggplot2); library(caret);
data(Wage); Wage <- subset(Wage, select=c(logwage))
## Get training/test sets
inTrain <- createDataPartition(y=Wage$wage, p=0.7, list=FALSE)
training <- Wage[inTrain,]; testing <- Wage[-inTrain,]
## Feature plot
featurePlot(x=training[,c("age","education","jobclass")], y=training$wage, plot="pairs
## Plot age vs. wage
qplot(age, wage, data=training)
## Plot age vs. wage, color by jobclass
qplot(age, wage, color=jobclass, data=training)
## Plot age vs. wage, color by education
qplot(age, wage, color=education, data=training)
## Fit a linear model with multiple variables
modFit <- train(wage ~ age + jobclass + education, method="lm", data=training)
finMod <- modFit$finalModel
## Diagnostics
plot(finMod, 1, pch=19, cex=0.5, col="#00000010")
## Color by variables not used in the model
qplot(finMod$fitted, finMod$residuals, color=race, data=training)
## Plot by index
plot(finMod$residuals, pch=19)
## Predicted vs. truth in test set
pred <- predict(modFit, testing)
qplot(wage, pred, color=year, data=testing)
## If you want to use all covariates
modFitAll <- train(wage ~ ., data=training, method="lm")
```

- Often useful in combination with other models.

Predicting with decision trees

- Key ideas:
  - Iteratively split variables into groups.
  - Evaluate homogeneity within each group.
  - Split again if necessary.
- Pros: easy to interpret, better performance in nonlinear settings.
- Cons: without pruning/cross-validation can lead to overfitting, harder to estimate uncertainty, results may be variable.
- Basic algorithm:
  - Start with all variables in one group.
  - Find the variable/split that best separates the outcomes.
  - Divide the data into two groups ("leaves") on that split/node.
  - Within each split, find the best variable/split that separates the outcomes.
  - Continue until the groups are too small or sufficiently pure to stop the algorithm.
- Measures of impurity:

    - In the mth leaf, there are N_m total objects that we might consider. You can count the number of times that class k appears in leaf m: $p_{mk} = \frac{1}{N_m} \sum_{x_i \text{ in leaf } m} \mathbb{I}(y_i = k)$
    - Misclassification error: $1 - p_{mk(m)}$where $k(m)$ is the most common class.
        * 0 = perfect purity.
        * 0.5 = no purity (no homogeneity).
    - Gini index: $1 - \sum_{k=1}^{K} p_{mk}^2$
        * 0 = perfect purity.
        * 0.5 = no purity.
    - Deviance (natural log)/information gain (log base 2): $-\sum_{k=1}^{K} \log_2 p_{mk}$
        * 0 = perfect purity.
        * 1 = no purity.

- Example: iris data

```
data(iris); library(ggplot2)
## Trying to predict species
table(iris$Species)
inTrain <- createDataPartition(y=iris$Species, p=0.7, list=F)
training <- iris[inTrain,]
testing <- iris[-inTrain,]
## Exploratory plot
qplot(Petal.Width, Sepal.Width, color=Species, data=training)
## Fit a model with rpart (one package for classification trees)
library(caret)
modFit <- train(Species ~ ., method="rpart", data=training)
print(modFit$finalModel)
## Plot classification tree
plot(modFit$finalModel, uniform=T, main="Classification Tree")
text(modFit$finalModel, use.n=T, all=T, cex=0.8)
## Another fancy plot
library(rattle)
fancyRpartPlot(modFit$finalModel)
## Predict new values
predict(modFit, newdata=testing)
```

- Notes:
    - Classification trees are non-linear models - they use interactions between variables.
    - Data transformations may be less important.
    - Trees can also be used for regression problems (continuous outcome).
    - Multiple tree building options in R both in the caret package (party, rpart) and outside (tree).

Bagging (bootstrap aggregating)

- When you fit complicated models, if you average them together, the resulting smoother fit gives a better balance between bias and variance in your fit.
- Basic idea:
    - Resample cases (with replacement) and recalculate predictions.
    - Average or majority vote.

- Notes:
    - Similar bias compared to any individual model.
    - Reduced variance.
    - Most useful for non-linear functions.

- Example: ozone data

```
library(ElemStatLearn); data(ozone, package="ElemStatLearn")
ozone <- ozone[order(ozone$ozone)]
head(ozone)
## Idea is to predict temperature as a function of ozone.
ll <- matrix(NA, nrow=10, ncol=155)
for (i in 1:10) {
        ## Subsample and re-order.
        ss <- sample(1:dim(ozone)[1], replace=T)
        ozone0 <- ozone[ss,]; ozone0 <- ozone0[order(ozone0$ozone),]
        ## Fit a Loess curve, span = measure of fit smoothness.
        loess0 <- loess(temperature ~ ozone, data=ozone0, span=0.2)
        ## Predict and store the results.
        ll[i,] <- predict(loess0, newdata=data.frame(ozone=1:155))
}
## Plot
plot(ozone$ozone, ozone$temperature, pch=19, cex=0.5
for (i in 1:10) {lines(1:155, ll[i,], col="gre", lwd=2)}
lines(1:155, apply(ll,2,mean), col="red", lwd=2)
```

- Bagging in caret: some models perform bagging for you, in the `train` function, consider `method` options.
    - bagEarth
    - treebag
    - bagFDA
    - Alternatively, you can bag any model you choose, using the `bag` function.

```
predictors <- data.frame(ozone=ozone$ozone)
temperature <- ozone$temperature
treebag <- bag(predictors, temperature, B=10,
                        bagControl = bagControl(fit = ctreeBag$fit,
                                                                        predict
                                                                        aggregat
```

- Parts of bagging:
    - Fit: takes in data frame and otucome that we passed and uses the `ctree` function to train a conditional regression tree.
    - Prediction: takes in the objects and a new dataset and gets a new prediction.
    - Aggregation: takes in those values and combines them in some way.
- Bagging is most useful for nonlinear models.
- Often used with trees, an extension is called “random forests.”
- Several models use bagging in caret’s `train` function.

Random forests

- Basic idea:
  - Bootstrap samples; rebuild classification and regression trees for each set of samples.
  - At each split, bootstrap variables - only a subset of variables is considered at each potential split..
  - Grow multiple trees and vote.
- Pros: accuracy.
- Cons: speed, interpretability, overfitting (very important to use cross-validation).
- Example: iris data.

```
data(iris); library(ggplot2)
inTrain <- createDataPartition(y=iris$species, p=0.7, list=F)
training <- iris[inTrain,]
testing <- iris[-inTrain,]
library(caret)
## Method rf is random forests.
modFit <- train(Species ~ ., data=training, method="rf", prox=T)
## Getting a single tree, k specifies which tree
getTree(modFit$finalModel, k=2)
## Class "centers"
irisP <- classCenter(training[,c(3,4)], training$Species, modFit$finalModel$prox)
irisP <- as.data.frame(irisP); irisP$Species <- rownames(irisP)
p <- qplot(Petal.Width, Petal.Length, col=Species, data=training)
p + geom_point(aes(x=Petal.Width,y=Petal.Length, col=Species), size=5, shape=4, data=i
## Predicting new values
pred <- predict(modFit, testing); testing$predRight <- pred==testing$Species
table(pred,testing$Species)
qplot(Petal.Width, Petal.Length, color=predRight, data=testing, main="newdata Predicti
```

- Random forests are usually one of the two top-performing algorithsm (along with boosting) in prediction contests.
- Random forests are difficult to interpret, but often very accurate.
- Care should be taken to avoid overfitting (see `rfcv` function).

Boosting

- Basic idea:
  - Take lots of (possibly) weak predictors.
  - Weight them and add them up.
  - Get a stronger predictor.
- Overview:
  - Start with a set of classifiers $h_1, ..., h_k$ (example: all possible trees, all possible regression models, etc.)
  - Create a classifier that combines classification functions: $f(x) = \text{sgn} \sum_{t=1}^{T} \alpha_t h_t(x)$
    - Goal is to minimize error on training set.
    - Iterative, select one h at each step.

    * Calculate weights based on errors.
    * Upweight missed classifications and select next h.

- Boosting in R:
  - Can be used with any subset of classifiers.
  - One large subclass is gradient boosting.
  - Multiple libraries in R: differences include the choice of basic classification functions and combination rules.
    * `gbm`: boosting with trees.
    * `mboost`: model-based boosting.
    * `ada`: statistical boosting based on additive logistic regression.
    * `gamBoost`: for boosting generalized additive models.
  - Most of these are available in the `caret` package.
- Example (wage data):

```
library(ISLR); data(Wage); library(ggplot2); library(caret);
Wage <- subset(Wage, select=-c(logwage))
inTrain <- createDataPartition(y=Wage$wage, p=0.7, list=F)
training <- Wage[inTrain,]; testing <- Wage[-inTrain,]
## Fit the model.
modFit <- train(wage ~ ., method="gbm", data=training, verbose=F)
print(modFit)
## Plot the results.
qplot(predict(modFit,testing), wage, data=testing)
```

Model-based prediction

- Basic idea:
  - Assume that the data follow a probabilistic model.
  - Use Bayes' theorem to identify optimal classifiers.
  - Pros: can take advantage of the structure of the data, may be computationally convenient, are reasonably accurate on real problems.
  - Cons: make additional assumptions about the data, and when the model is incorrect, you may get reduced accuracy.
- Model based approach:
  - Goal is to build a parametric model for conditional distribution $P(Y = k|X = x)$ (probabily that our outcome Y is in some class k given our predictor variables X in a class x.
  - A typical approach is to apply Bayes' theorem:
    * $P(Y = k|X = x) = \frac{P(X=x|Y=k)P(Y=k)}{\sum_{l=1}^{K} P(X=x|Y=l)P(Y=l)}$
    * $P(Y = k|X = x) = \frac{f_k(x)\pi_k}{\sum_{l=1}^{K} f_l(x)\pi_l}$
  - Typically, prior probabilities π_k are set in advance.
  - A common choice for $f_k(x)$ is a Gaussian distribution.
  - Estimate the parameters $(\mu_{k,}\sigma_k^2)$ from the data.
  - Classify to the class with the highest value of $P(Y = k|X = x)$.

- Classifying using the model: a range of models use this approach.
  - Linear discriminant analysis assumes $f_k(x)$ is multivariate Gaussian with the same covariances.
    - Discriminant function: $\delta_k(x) = x^T \Sigma^{-1} \mu_k - \frac{1}{2} \mu_k \Sigma^{-1} \mu_k + \log(\mu_k)$
    - Decide on class based on $\hat{Y}(x) = \text{argmax}\ (x_k, \delta_k(x))$.
    - We usually estimate parameters with maximum likelihood.
  - Quadratic discriminant analysis assumes $f_k(x)$ is multivariate Gaussian with different covariances.
  - Model based prediction assumes more complicated versions for the covariance matrix.
  - Naive Bayes assumes independence between features for model building.
    - Suppose we have many predictors we would want to model $P(Y = k | X_1, ..., X_m)$.
    - We could use Bayes' theorem to get $P(Y = k | X_1, ..., X_m) = \frac{\pi_k P(X_1, ..., X_m | Y = k)}{\sum_{l=1}^{K} P(X_1, ..., X_m | Y = k) \pi_l} \propto \pi_k P(X_1, ..., X_m | Y = k)$.
    - This can be written as $\pi_k P(X_1 | Y = k) P(X_2 | X_1, Y = k) ... P(X_m | X_1, ..., X_{m-1} | Y = k)$.
    - We could make an assumption of independence to write: $\approx \pi_k P(X_1 | Y = k) P(X_2 | Y = k) ... P(X_m | Y = k)$.
      - This is not a great assumption always, hence the name "naive" Bayes.
- Example: iris data

```
data(iris); library(ggplot2)
names(iris)
## Make training and testing sets
inTrain <- createDataPartition(y=iris$Species, p=0.7, list=F)
training <- iris[inTrain,]
testing <- iris[-inTrain,]
## Build predictions with LDA and NB.
modlda <- train(Species ~ ., data=trianing, method="lda")
modnb <- train(Species ~ ., data=training, method="nb")
plda <- predict(modlda, testing); pnb <- predict(modnb, testing)
## Compare results
table(plda,pnb)
equalPredictions <- (plda == pnb)
qplot(Petal.Width, Sepal.Width, color=equalPredictions, data=testing)
```

Regularized regression

- Basic idea: fit a regression model, then penalize or shrink large coefficients.
- Pros: can help with the bias/variance tradeoff and can help with model selection.
- Cons: may be computationally demanding, does not perform as well as random forests or boosting.
- A motivating example:
  - Fit a regression example $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \epsilon$
  - If X_1 and X_2 are nearly perfectly correlated (co-linear), you can approximate the model by $Y = \beta_0 + (\beta_1 + \beta_2) X_1 + \epsilon$.
  - The result is:
    - You will get a good estimate of Y.
    - The estimate of Y will be biased.
    - We may reduce variance in the estimate.
- A common pattern is that training error will always go down as we add predictors. The testing error will go down to a certain point, but then it will start going back up, due to overfitting.

- Model selection approach: split samples
  - No better method when data/computation time permits it.
  - Approach:
    - Divide data into training/test/validation.
    - Treat validation as test data, train all competing models on the train data and pick the best one on validation.
    - To appropriately assess performance on new data, apply it to the test set.
    - You may re-spilt and re-perform steps 1-3.
  - Two common problems:
    - Limited data, may not be able to do many subsets of data.
    - Computational complexity.
- Decomposing expected prediction error
  - Assume $Y_i = f(X_i) + \epsilon_i$
  - Expected prediction error: $EPE(\lambda) = E\left[\left\{Y - \hat{f}_\lambda(X)\right\}^2\right]$
  - Suppose $\hat{f}_\lambda$ is the estimate from the training data and look at a new data point $X = x^\star$:
    - $E\left[\left\{Y - \hat{f}_\lambda(x^\star)\right\}^2\right] = \sigma^2 + \left\{E\left[\hat{f}_\lambda(x^\star)\right] - f(x^\star)\right\}^2 + Var\left[\hat{f}_\lambda(x_0)\right] = \text{Irreducible error} + \text{Bias}^2 + \text{Variance}$
- Hard thresholding
  - Model $Y = f(X) + \epsilon$
  - Set $\hat{f}_\lambda(x) = x'\beta$
  - Constrain only λ coefficients to be nonzero.
  - Selection problem is after choosing λ, figure out which $p - \lambda$ coefficients to make nonzero.
- Regularization for regression
  - If the β_js are unconstrained, they can explode. Hence, they are susceptible to very high variance.
  - To control variance, we might regularize/shrink the coefficients:
    - $PRSS(\beta) = \sum_{j=1}^n \left(Y_j - \sum_{i=1}^m B_{1i} X_{ij}\right)^2 + P(\lambda; \beta)$, where PRSS is a penalized form of the sum of squares.
  - Things that are commonly looked for:
    - Penalty reduces complexity.
    - Penalty reduces variance.
    - Penalty respects structure of the problem.
- Ridge regression
  - Solve: $\sum_{i=1}^N \left(y_i - \beta_0 + \sum_{j=1}^p x_{ij}\beta_j\right)^2 + \lambda \sum_{j=1}^p \beta_j^2$.
  - This is equivalent to solving $\sum_{i=1}^N \left(y_i - \beta_0 + \sum_{j=1}^p x_{ij}\beta_j\right)^2$ subject to $\sum_{j=1}^p \beta_j^2 \leq s$, where s is inversely proportional to λ.
  - Inclusion of λ makes the problem non-singular even if $X^T X$ is not invertible.
- Tuning parameter λ

- Controls the size of the coefficients.
- Controls the amount of regularization.
- As $\lambda \to 0$, we obtain the least squares solution.
- As $\lambda \to \infty$, we have $\hat{\beta}^{ridge}_{\lambda=\infty} = 0$.

- In `caret` package, some methods for fitting penalized regression models are:
  - `ridge`
  - `lasso`
  - `relaxo`

Combining predictors

- Also known as ensembling methods.
- Key ideas:
  - Combine classifiers by averaging or voting. In general these can be very different classifiers.
  - Combing classifers improves accuracy, but reduces interpretability.
  - Boosting, bagging, and random forests are variants on this theme, but they all average the same kinds of classifiers.
- Example: Netflix prize "BellKor" combined 107 predictors.
- Basic intuition - majority vote.
  - If we have 5 completely independent classifiers, and the accuracy is 70% for each, $10 \times 0.7^3 \times 0.3^2 + 5 \times 0.7^4 \times 0.3 + 0.7^5 = 0.837$ majority vote accuracy.
  - With 101 independent classifiers, we get 0.999 majority vote accuracy.
- Approaches for combining classifiers:
  - Bagging, boosting, random forests - these all usually combine similar classifiers.
  - Combining different classifiers - model stacking and model ensembling.
- Model stacking - example with Wage data.

```
library(ISLR); data(Wage); library(ggplot2); library(caret);
Wage <- subset(Wage, select=-c(logwage))
## Create a building data set and a validation data set.
inBuild <- createDataPartition(y=Wage$wage, p=0.7, list=F)
validation <- Wage[-inBuild,]; buildData <- Wage[inBuild,]
## Split build data into training and testing.
inTrain <- createDataPartition(y=buildData$wage, p=0.7, list=F)
training <- buildData[inTrain,]; testing <- buildData[-inTrain,]
## Build two different models with the training set.
mod1 <- train(wage ~ ., method="glm", data=training)
mod2 <- train(wage ~ ., method="rf", data=training, trControl = trainControl(method="c
## Predict on the testing set.
pred1 <- predict(mod1, testing); pred2 <- predict(mod2, testing)
qplot(pred1,pred2,color=wage,data=testing)
## Fit a model that combines predictors.
predDF <- data.frame(pred1, pred2, wage=testing$wage)
combModFit <- train(wage ~ ., method="gam", data=predDF)
```

```
combPred <- predict(combModFit, predDF)
## Testing errors.
sqrt(sum((pred1-testing$wage)^2))
sqrt(sum((pred2-testing$wage)^2))
sqrt(sum((combPred-testing$wage)^2))
## Predict on validation data set.
pred1V <- predict(mod1, validation); pred2V <- predict(mod2, validation)
predVDF <- data.frame(pred1=pred1V, pred2=pred2V)
combPredV <- predict(combModFit, predVDF)
## Evaluate on validation.
sqrt(sum((pred1V-validation$wage)^2))
sqrt(sum((pred2V-validation$wage)^2))
sqrt(sum((combPredV-validation$wage)^2))
```

- Even simple blending can be useful to improve accuracy.
- Typical model for binary/multiclass data.
  - Build an odd number of models.
  - Predict with each model.
  - Predict the class by majority vote.
- This can get much more complicated:
  - Simple blending in caret: `caretEnsemble` (use at your own risk).
- Recall: scalability matters! This can be very computationally complex, hard to scale up to large data sets.

Forecasting

- Forecasting is a type of prediction problem that applies to time-series data (stocks, for example).
- What is different?
  - Data are time-dependent.
  - Specific pattern types:
    - Trends - long-term increase or decrease.
    - Seasonal patterns - related to time of week, month, year, etc. There is a pattern which recurs over a fixed period of time.
    - Cycles - patterns that rise and fall periodically over non-fixed periods of time.
  - Subsampling into training/test sets is more complicated.
  - Similar issues arise in spatial data:
    - Dependency between nearby observations.
    - Location-specific effects.
  - Typically, the goal is to predict one or more observations into the future.
  - All standard predictions can be used (with caution)!
- Also common in geographic analyses.
- Beware extrapolation!
- Useful for forecasting: simple moving average.

- $Y_t = \frac{1}{2k+1} \sum_{j=-k}^{k} y_{t+j}$

- Also useful: exponential smoothing - weighting nearby points in time more heavily than those which are farther away.
  - $\hat{y}_{t+1} = \alpha y_t + (1-\alpha)\,\hat{y}_{t-1}$
- Example: Google data

```
library(quantmod)
from.dat <- as.Date("01/01/08", format"%m/%d/%y")
to.dat <- as.Date("12/31/13", format="%m/%d/%y")
getSymbols("GOOG", src="google", from = from.dat, to = to.dat)
## Get monthly summary and store as a time-series.
mGoog <- to.monthly(GOOG)
googOpen <- Op(mGoog)
ts1 <- ts(googOpen, frequency=12)
plot(ts1, xlab="Years+1", ylab="GOOG")
## Decompose a time-series into parts (trends, patterns, cycles).
plot(decompose(ts1), xlab="Years+1")
## Training and test sets.
ts1Train <- window(ts1, start=1, end=5)
ts1Test <- window(ts1, start=5, end=(7-0.01))
## Simple moving average.
plot(tst1Train)
lines(ma(ts1Train, order=3), col="red")
## Exponential smoothing.
ets1 <- ets(ts1Train, model="MMM")
fcast <- forecast(ets1)
plot(fcast); lines(ts1Test, col="red")
## Get the accuracy.
accuracy(fcast, ts1Test)
```

- Forecasting and time-series prediction is an entire field.
- Rob Hyndman's "Forecasting: principles and practice" is a good place to start. (free online book)
- Cautions:
  - Be wary of spurious correlations.
  - Be careful about how far you predict into the future.
  - Be wary of dependencies over time.
- See the `quantmod` or `quandl` packages for finance-related problems.

Unsupervised prediction

- Key ideas:
  - Sometimes you don't know the labels for prediction.
  - To build a predictor:
    * Create clusters that you're observed (not always obvious).
    * Add names to the clusters (i.e., how to interpret the clusters).
    * Build a predictor for clusters.

- In a new dataset, predict clusters.

- Example: iris data ignoring species clusters.

```
data(iris); library(ggplot2)
inTrain <- createDataPartition(y=iris#Species, p=0.7, list=F)
training <- iris[inTrain,]
testing <- iris[-inTrain,]
## Clustering with k-means
kMeans1 <- kmeans(subset(training, select=-c(Species)), centers=3)
training$clusters <- as.factor(kMeans1$cluster)
qplot(Petal.Width, Petal.Length, color=clusters, data=training)
## Compare to real labels.
table(kMean1$cluster, training$Species)
## Build a predictor.
modFit <- train(clusters ~ ., data=subset(training, select=-c(Species)), method="rpart
table(predict(modFit,training), training$Species)
## Apply to test dataset.
testClusterPred <- predict(modFit, testing)
table(testClusterPred, testing$Species)
```

- Notes:
  - The `cl_predict` function in the `clue` package provides similar functionality.
  - Beware over-interpretation of clusters!
  - This is one basic approach to recommendation engines.